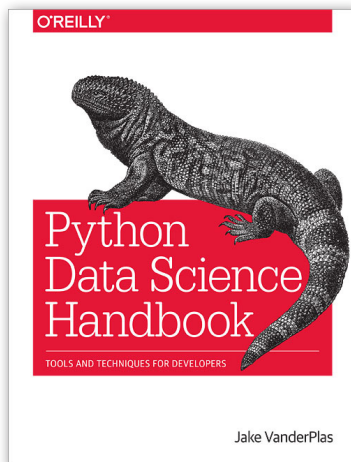
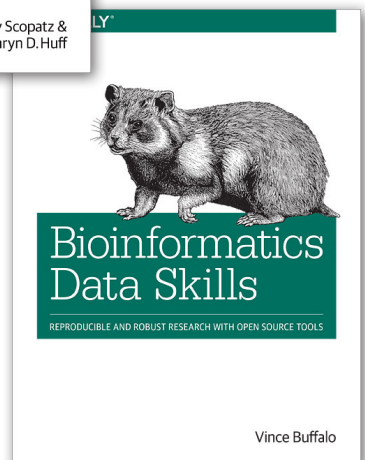
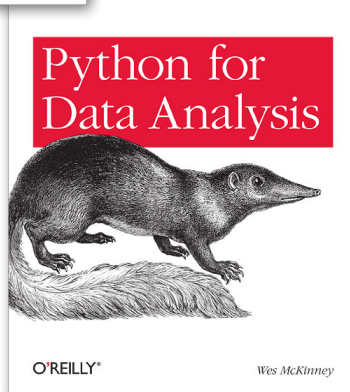
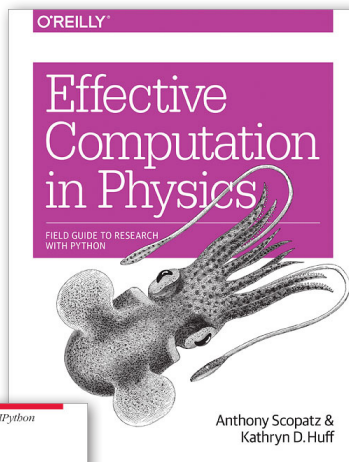


Python for Scientists

A Curated Collection of Chapters from the
O'Reilly Data and Programming Library



Wrangling with Pandas, NumPy, and IPython



Python for Scientists

A Curated Collection of Chapters from the O'Reilly Data and Programming Library

More and more, scientists are seeing tech seep into their work. From data collection to team management, various tools exist to make your lives easier. But, where to start? Python is growing in popularity in scientific circles, due to its simple syntax and seemingly endless libraries. This free ebook gets you started on the path to a more streamlined process. With a collection of chapters from our top scientific books, you'll learn about the various options that await you as you strengthen your computational thinking.

For more information on current & forthcoming Programming content, check out www.oreilly.com/programming/free/.



Python for Data Analysis

[Available here](#)

Python Language Essentials Appendix

Effective Computation in Physics

[Available here](#)

Chapter 1: Introduction to the Command Line

Chapter 7: Analysis and Visualization

Chapter 20: Publication

Bioinformatics Data Skills

[Available here](#)

Chapter 4: Working with Remote Machines

Chapter 5: Git for Scientists

Python Data Science Handbook

[Available here](#)

Chapter 3: Introduction to NumPy

Chapter 4: Introduction to Pandas

Data Wrangling with Pandas, NumPy, and IPython

Python for Data Analysis



O'REILLY®

Wes McKinney

Python for Data Analysis

Wes McKinney

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Python Language Essentials

Knowledge is a treasure, but practice is the key to it.

—Thomas Fuller

People often ask me about good resources for learning Python for data-centric applications. While there are many excellent Python language books, I am usually hesitant to recommend some of them as they are intended for a general audience rather than tailored for someone who wants to load in some data sets, do some computations, and plot some of the results. There are actually a couple of books on “scientific programming in Python”, but they are geared toward numerical computing and engineering applications: solving differential equations, computing integrals, doing Monte Carlo simulations, and various topics that are more mathematically-oriented rather than being about data analysis and statistics. As this is a book about becoming proficient at working with data in Python, I think it is valuable to spend some time highlighting the most important features of Python’s built-in data structures and libraries from the perspective of processing and manipulating structured and unstructured data. As such, I will only present roughly enough information to enable you to follow along with the rest of the book.

This chapter is not intended to be an exhaustive introduction to the Python language but rather a biased, no-frills overview of features which are used repeatedly throughout this book. For new Python programmers, I recommend that you supplement this chapter with the official Python tutorial (<http://docs.python.org>) and potentially one of the many excellent (and much longer) books on general purpose Python programming. In my opinion, it is *not* necessary to become proficient at building good software in Python to be able to productively do data analysis. I encourage you to use IPython to experiment with the code examples and to explore the documentation for the various types, functions, and methods. Note that some of the code used in the examples may not necessarily be fully-introduced at this point.

Much of this book focuses on high performance array-based computing tools for working with large data sets. In order to use those tools you must often first do some munging to corral messy data into a more nicely structured form. Fortunately, Python is one of

the easiest-to-use languages for rapidly whipping your data into shape. The greater your facility with Python, the language, the easier it will be for you to prepare new data sets for analysis.

The Python Interpreter

Python is an *interpreted* language. The Python interpreter runs a program by executing one statement at a time. The standard interactive Python interpreter can be invoked on the command line with the `python` command:

```
$ python
Python 2.7.2 (default, Oct 4 2011, 20:06:09)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 5
>>> print a
5
```

The `>>>` you see is the *prompt* where you'll type expressions. To exit the Python interpreter and return to the command prompt, you can either type `exit()` or press `Ctrl-D`.

Running Python programs is as simple as calling `python` with a `.py` file as its first argument. Suppose we had created `hello_world.py` with these contents:

```
print 'Hello world'
```

This can be run from the terminal simply as:

```
$ python hello_world.py
Hello world
```

While many Python programmers execute all of their Python code in this way, many *scientific* Python programmers make use of IPython, an enhanced interactive Python interpreter. [Chapter 3](#) is dedicated to the IPython system. By using the `%run` command, IPython executes the code in the specified file in the same process, enabling you to explore the results interactively when it's done.

```
$ ipython
Python 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul 3 2011, 15:17:51)
Type "copyright", "credits" or "license" for more information.

IPython 0.12 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

In [1]: %run hello_world.py
Hello world

In [2]:
```

The default IPython prompt adopts the numbered `In [2]:` style compared with the standard `>>>` prompt.

The Basics

Language Semantics

The Python language design is distinguished by its emphasis on readability, simplicity, and explicitness. Some people go so far as to liken it to “executable pseudocode”.

Indentation, not braces

Python uses whitespace (tabs or spaces) to structure code instead of using braces as in many other languages like R, C++, Java, and Perl. Take the for loop in the above quicksort algorithm:

```
for x in array:
    if x < pivot:
        less.append(x)
    else:
        greater.append(x)
```

A colon denotes the start of an indented code block after which all of the code must be indented by the same amount until the end of the block. In another language, you might instead have something like:

```
for x in array {
    if x < pivot {
        less.append(x)
    } else {
        greater.append(x)
    }
}
```

One major reason that whitespace matters is that it results in most Python code looking cosmetically similar, which means less cognitive dissonance when you read a piece of code that you didn't write yourself (or wrote in a hurry a year ago!). In a language without significant whitespace, you might stumble on some differently formatted code like:

```
for x in array
{
    if x < pivot
    {
        less.append(x)
    }
    else
    {
        greater.append(x)
    }
}
```

```
}  
}
```

Love it or hate it, significant whitespace is a fact of life for Python programmers, and in my experience it helps make Python code a lot more readable than other languages I've used. While it may seem foreign at first, I suspect that it will grow on you after a while.



I strongly recommend that you use *4 spaces* to as your default indentation and that your editor replace tabs with 4 spaces. Many text editors have a setting that will replace tab stops with spaces automatically (do this!). Some people use tabs or a different number of spaces, with 2 spaces not being terribly uncommon. 4 spaces is by and large the standard adopted by the vast majority of Python programmers, so I recommend doing that in the absence of a compelling reason otherwise.

As you can see by now, Python statements also do not need to be terminated by semicolons. Semicolons can be used, however, to separate multiple statements on a single line:

```
a = 5; b = 6; c = 7
```

Putting multiple statements on one line is generally discouraged in Python as it often makes code less readable.

Everything is an object

An important characteristic of the Python language is the consistency of its *object model*. Every number, string, data structure, function, class, module, and so on exists in the Python interpreter in its own “box” which is referred to as a *Python object*. Each object has an associated *type* (for example, *string* or *function*) and internal data. In practice this makes the language very flexible, as even functions can be treated just like any other object.

Comments

Any text preceded by the hash mark (pound sign) `#` is ignored by the Python interpreter. This is often used to add comments to code. At times you may also want to exclude certain blocks of code without deleting them. An easy solution is to *comment out* the code:

```
results = []  
for line in file_handle:  
    # keep the empty lines for now  
    # if len(line) == 0:  
    #     continue  
    results.append(line.replace('foo', 'bar'))
```


Function and object method calls

Functions are called using parentheses and passing zero or more arguments, optionally assigning the returned value to a variable:

```
result = f(x, y, z)
g()
```

Almost every object in Python has attached functions, known as *methods*, that have access to the object's internal contents. They can be called using the syntax:

```
obj.some_method(x, y, z)
```

Functions can take both *positional* and *keyword* arguments:

```
result = f(a, b, c, d=5, e='foo')
```

More on this later.

Variables and pass-by-reference

When assigning a variable (or *name*) in Python, you are creating a *reference* to the object on the right hand side of the equals sign. In practical terms, consider a list of integers:

```
In [241]: a = [1, 2, 3]
```

Suppose we assign *a* to a new variable *b*:

```
In [242]: b = a
```

In some languages, this assignment would cause the data `[1, 2, 3]` to be copied. In Python, *a* and *b* actually now refer to the same object, the original list `[1, 2, 3]` (see [Figure A-1](#) for a mockup). You can prove this to yourself by appending an element to *a* and then examining *b*:

```
In [243]: a.append(4)
```

```
In [244]: b
Out[244]: [1, 2, 3, 4]
```

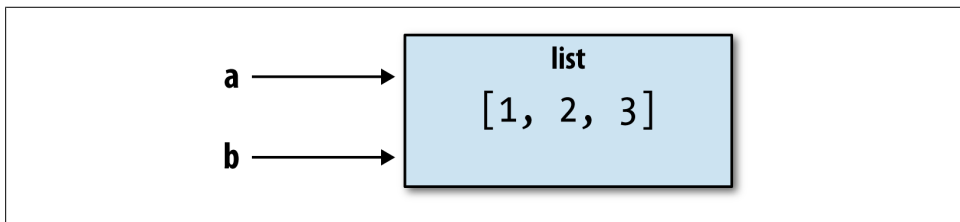


Figure A-1. Two references for the same object

Understanding the semantics of references in Python and when, how, and why data is copied is especially critical when working with larger data sets in Python.



Assignment is also referred to as *binding*, as we are binding a name to an object. Variable names that have been assigned may occasionally be referred to as bound variables.

When you pass objects as arguments to a function, you are only passing references; no copying occurs. Thus, Python is said to *pass by reference*, whereas some other languages support both pass by value (creating copies) and pass by reference. This means that a function can mutate the internals of its arguments. Suppose we had the following function:

```
def append_element(some_list, element):
    some_list.append(element)
```

Then given what's been said, this should not come as a surprise:

```
In [2]: data = [1, 2, 3]

In [3]: append_element(data, 4)

In [4]: data
Out[4]: [1, 2, 3, 4]
```

Dynamic references, strong types

In contrast with many compiled languages, such as Java and C++, object *references* in Python have no type associated with them. There is no problem with the following:

```
In [245]: a = 5           In [246]: type(a)
                                Out[246]: int

In [247]: a = 'foo'      In [248]: type(a)
                                Out[248]: str
```

Variables are names for objects within a particular namespace; the type information is stored in the object itself. Some observers might hastily conclude that Python is not a “typed language”. This is not true; consider this example:

```
In [249]: '5' + 5
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-249-f9dbf5f0b234> in <module>()
----> 1 '5' + 5
TypeError: cannot concatenate 'str' and 'int' objects
```

In some languages, such as Visual Basic, the string '5' might get implicitly converted (or *casted*) to an integer, thus yielding 10. Yet in other languages, such as JavaScript, the integer 5 might be casted to a string, yielding the concatenated string '55'. In this regard Python is considered a *strongly-typed* language, which means that every object has a specific type (or *class*), and implicit conversions will occur only in certain obvious circumstances, such as the following:

```

In [250]: a = 4.5

In [251]: b = 2

# String formatting, to be visited later
In [252]: print 'a is %s, b is %s' % (type(a), type(b))
a is <type 'float'>, b is <type 'int'>

In [253]: a / b
Out[253]: 2.25

```

Knowing the type of an object is important, and it's useful to be able to write functions that can handle many different kinds of input. You can check that an object is an instance of a particular type using the `isinstance` function:

```

In [254]: a = 5
In [255]: isinstance(a, int)
Out[255]: True

```

`isinstance` can accept a tuple of types if you want to check that an object's type is among those present in the tuple:

```

In [256]: a = 5; b = 4.5

In [257]: isinstance(a, (int, float))
Out[257]: True

In [258]: isinstance(b, (int, float))
Out[258]: True

```

Attributes and methods

Objects in Python typically have both attributes, other Python objects stored “inside” the object, and methods, functions associated with an object which can have access to the object's internal data. Both of them are accessed via the syntax `obj.attribute_name`:

```

In [1]: a = 'foo'

In [2]: a.<Tab>
a.capitalize a.format      a.isupper    a.rindex     a.strip
a.center     a.index      a.join       a.rjust      a.swapcase
a.count      a.isalnum   a.ljust      a.rpartition a.title
a.decode     a.isalpha   a.lower      a.rsplit     a.translate
a.encode     a.isdigit   a.lstrip     a.rstrip     a.upper
a.endswith   a.islower   a.partition  a.split      a.zfill
a.expandtabs a.ispace    a.replace    a.splitlines
a.find       a.istitle   a.rfind      a.startswith

```

Attributes and methods can also be accessed by name using the `getattr` function:

```

>>> getattr(a, 'split')
<function split>

```

While we will not extensively use the functions `getattr` and related functions `hasattr` and `setattr` in this book, they can be used very effectively to write generic, reusable code.

“Duck” typing

Often you may not care about the type of an object but rather only whether it has certain methods or behavior. For example, you can verify that an object is iterable if it implemented the *iterator protocol*. For many objects, this means it has a `__iter__` “magic method”, though an alternative and better way to check is to try using the `iter` function:

```
def isiterable(obj):
    try:
        iter(obj)
        return True
    except TypeError: # not iterable
        return False
```

This function would return `True` for strings as well as most Python collection types:

```
In [260]: isiterable('a string')      In [261]: isiterable([1, 2, 3])
Out[260]: True                        Out[261]: True

In [262]: isiterable(5)
Out[262]: False
```

A place where I use this functionality all the time is to write functions that can accept multiple kinds of input. A common case is writing a function that can accept any kind of sequence (list, tuple, ndarray) or even an iterator. You can first check if the object is a list (or a NumPy array) and, if it is not, convert it to be one:

```
if not isinstance(x, list) and isiterable(x):
    x = list(x)
```

Imports

In Python a *module* is simply a `.py` file containing function and variable definitions along with such things imported from other `.py` files. Suppose that we had the following module:

```
# some_module.py
PI = 3.14159

def f(x):
    return x + 2

def g(a, b):
    return a + b
```

If we wanted to access the variables and functions defined in `some_module.py`, from another file in the same directory we could do:

```
import some_module
result = some_module.f(5)
pi = some_module.PI
```

Or equivalently:

```
from some_module import f, g, PI
result = g(5, PI)
```

By using the `as` keyword you can give imports different variable names:

```
import some_module as sm
from some_module import PI as pi, g as gf

r1 = sm.f(pi)
r2 = gf(6, pi)
```

Binary operators and comparisons

Most of the binary math operations and comparisons are as you might expect:

```
In [263]: 5 - 7          In [264]: 12 + 21.5
Out[263]: -2            Out[264]: 33.5

In [265]: 5 <= 2
Out[265]: False
```

See [Table A-1](#) for all of the available binary operators.

To check if two references refer to the same object, use the `is` keyword. `is not` is also perfectly valid if you want to check that two objects are not the same:

```
In [266]: a = [1, 2, 3]

In [267]: b = a

# Note, the list function always creates a new list
In [268]: c = list(a)

In [269]: a is b          In [270]: a is not c
Out[269]: True           Out[270]: True
```

Note this is not the same thing as comparing with `==`, because in this case we have:

```
In [271]: a == c
Out[271]: True
```

A very common use of `is` and `is not` is to check if a variable is `None`, since there is only one instance of `None`:

```
In [272]: a = None

In [273]: a is None
Out[273]: True
```

Table A-1. Binary operators

Operation	Description
<code>a + b</code>	Add a and b
<code>a - b</code>	Subtract b from a
<code>a * b</code>	Multiply a by b
<code>a / b</code>	Divide a by b
<code>a // b</code>	Floor-divide a by b, dropping any fractional remainder

Operation	Description
<code>a ** b</code>	Raise <code>a</code> to the <code>b</code> power
<code>a & b</code>	True if both <code>a</code> and <code>b</code> are True. For integers, take the bitwise AND.
<code>a b</code>	True if either <code>a</code> or <code>b</code> is True. For integers, take the bitwise OR.
<code>a ^ b</code>	For booleans, True if <code>a</code> or <code>b</code> is True, but not both. For integers, take the bitwise EXCLUSIVE-OR.
<code>a == b</code>	True if <code>a</code> equals <code>b</code>
<code>a != b</code>	True if <code>a</code> is not equal to <code>b</code>
<code>a <= b</code> , <code>a < b</code>	True if <code>a</code> is less than (less than or equal) to <code>b</code>
<code>a > b</code> , <code>a >= b</code>	True if <code>a</code> is greater than (greater than or equal) to <code>b</code>
<code>a is b</code>	True if <code>a</code> and <code>b</code> reference same Python object
<code>a is not b</code>	True if <code>a</code> and <code>b</code> reference different Python objects

Strictness versus laziness

When using any programming language, it's important to understand *when* expressions are evaluated. Consider the simple expression:

```
a = b = c = 5
d = a + b * c
```

In Python, once these statements are evaluated, the calculation is immediately (or *strictly*) carried out, setting the value of `d` to 30. In another programming paradigm, such as in a pure functional programming language like Haskell, the value of `d` might not be evaluated until it is actually used elsewhere. The idea of deferring computations in this way is commonly known as *lazy evaluation*. Python, on the other hand, is a very *strict* (or eager) language. Nearly all of the time, computations and expressions are evaluated immediately. Even in the above simple expression, the result of `b * c` is computed as a separate step before adding it to `a`.

There are Python techniques, especially using iterators and generators, which can be used to achieve laziness. When performing very expensive computations which are only necessary some of the time, this can be an important technique in data-intensive applications.

Mutable and immutable objects

Most objects in Python are mutable, such as lists, dicts, NumPy arrays, or most user-defined types (classes). This means that the object or values that they contain can be modified.

```
In [274]: a_list = ['foo', 2, [4, 5]]
```

```
In [275]: a_list[2] = (3, 4)
```

```
In [276]: a_list
Out[276]: ['foo', 2, (3, 4)]
```

Others, like strings and tuples, are immutable:

```
In [277]: a_tuple = (3, 5, (4, 5))

In [278]: a_tuple[1] = 'four'
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-278-b7966a9ae0f1> in <module>()
----> 1 a_tuple[1] = 'four'
TypeError: 'tuple' object does not support item assignment
```

Remember that just because you *can* mutate an object does not mean that you always *should*. Such actions are known in programming as *side effects*. For example, when writing a function, any side effects should be explicitly communicated to the user in the function’s documentation or comments. If possible, I recommend trying to avoid side effects and *favor immutability*, even though there may be mutable objects involved.

Scalar Types

Python has a small set of built-in types for handling numerical data, strings, boolean (True or False) values, and dates and time. See [Table A-2](#) for a list of the main scalar types. Date and time handling will be discussed separately as these are provided by the `datetime` module in the standard library.

Table A-2. Standard Python Scalar Types

Type	Description
None	The Python “null” value (only one instance of the None object exists)
str	String type. ASCII-valued only in Python 2.x and Unicode in Python 3
unicode	Unicode string type
float	Double-precision (64-bit) floating point number. Note there is no separate double type.
bool	A True or False value
int	Signed integer with maximum value determined by the platform.
long	Arbitrary precision signed integer. Large int values are automatically converted to long.

Numeric types

The primary Python types for numbers are `int` and `float`. The size of the integer which can be stored as an `int` is dependent on your platform (whether 32 or 64-bit), but Python will transparently convert a very large integer to `long`, which can store arbitrarily large integers.

```
In [279]: ival = 17239871

In [280]: ival ** 6
Out[280]: 26254519291092456596965462913230729701102721L
```

Floating point numbers are represented with the Python `float` type. Under the hood each one is a double-precision (64 bits) value. They can also be expressed using scientific notation:

```
In [281]: fval = 7.243
```

```
In [282]: fval2 = 6.78e-5
```

In Python 3, integer division not resulting in a whole number will always yield a floating point number:

```
In [284]: 3 / 2
Out[284]: 1.5
```

In Python 2.7 and below (which some readers will likely be using), you can enable this behavior by default by putting the following cryptic-looking statement at the top of your module:

```
from __future__ import division
```

Without this in place, you can always explicitly convert the denominator into a floating point number:

```
In [285]: 3 / float(2)
Out[285]: 1.5
```

To get C-style integer division (which drops the fractional part if the result is not a whole number), use the floor division operator `//`:

```
In [286]: 3 // 2
Out[286]: 1
```

Complex numbers are written using `j` for the imaginary part:

```
In [287]: cval = 1 + 2j

In [288]: cval * (1 - 2j)
Out[288]: (5+0j)
```

Strings

Many people use Python for its powerful and flexible built-in string processing capabilities. You can write *string literal* using either single quotes `'` or double quotes `"`:

```
a = 'one way of writing a string'
b = "another way"
```

For multiline strings with line breaks, you can use triple quotes, either `'''` or `"""`:

```
c = """
This is a longer string that
spans multiple lines
"""
```

Python strings are immutable; you cannot modify a string without creating a new string:

```

In [289]: a = 'this is a string'

In [290]: a[10] = 'f'
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-290-5ca625d1e504> in <module>()
----> 1 a[10] = 'f'
TypeError: 'str' object does not support item assignment

In [291]: b = a.replace('string', 'longer string')

In [292]: b
Out[292]: 'this is a longer string'

```

Many Python objects can be converted to a string using the `str` function:

```

In [293]: a = 5.6          In [294]: s = str(a)

In [295]: s
Out[295]: '5.6'

```

Strings are a sequence of characters and therefore can be treated like other sequences, such as lists and tuples:

```

In [296]: s = 'python'      In [297]: list(s)
Out[297]: ['p', 'y', 't', 'h', 'o', 'n']

In [298]: s[:3]
Out[298]: 'pyt'

```

The backslash character `\` is an *escape character*, meaning that it is used to specify special characters like newline `\n` or unicode characters. To write a string literal with backslashes, you need to escape them:

```

In [299]: s = '12\\34'

In [300]: print s
12\34

```

If you have a string with a lot of backslashes and no special characters, you might find this a bit annoying. Fortunately you can preface the leading quote of the string with `r` which means that the characters should be interpreted as is:

```

In [301]: s = r'this\has\no\special\characters'

In [302]: s
Out[302]: 'this\\has\\no\\special\\characters'

```

Adding two strings together concatenates them and produces a new string:

```

In [303]: a = 'this is the first half '

In [304]: b = 'and this is the second half'

In [305]: a + b
Out[305]: 'this is the first half and this is the second half'

```

String templating or formatting is another important topic. The number of ways to do so has expanded with the advent of Python 3, here I will briefly describe the mechanics of one of the main interfaces. Strings with a % followed by one or more format characters is a target for inserting a value into that string (this is quite similar to the `printf` function in C). As an example, consider this string:

```
In [306]: template = '%.2f %s are worth $%d'
```

In this string, %s means to format an argument as a string, %.2f a number with 2 decimal places, and %d an integer. To substitute arguments for these format parameters, use the binary operator % with a tuple of values:

```
In [307]: template % (4.5560, 'Argentine Pesos', 1)
Out[307]: '4.56 Argentine Pesos are worth $1'
```

String formatting is a broad topic; there are multiple methods and numerous options and tweaks available to control how values are formatted in the resulting string. To learn more, I recommend you seek out more information on the web.

I discuss general string processing as it relates to data analysis in more detail in [Chapter 7](#).

Booleans

The two boolean values in Python are written as `True` and `False`. Comparisons and other conditional expressions evaluate to either `True` or `False`. Boolean values are combined with the `and` and `or` keywords:

```
In [308]: True and True
Out[308]: True
```

```
In [309]: False or True
Out[309]: True
```

Almost all built-in Python types and any class defining the `__nonzero__` magic method have a `True` or `False` interpretation in an `if` statement:

```
In [310]: a = [1, 2, 3]
.....: if a:
.....:     print 'I found something!'
.....:
I found something!

In [311]: b = []
.....: if not b:
.....:     print 'Empty!'
.....:
Empty!
```

Most objects in Python have a notion of true- or falseness. For example, empty sequences (lists, dicts, tuples, etc.) are treated as `False` if used in control flow (as above with the empty list `b`). You can see exactly what boolean value an object coerces to by invoking `bool` on it:


```
In [312]: bool([]), bool([1, 2, 3])
Out[312]: (False, True)

In [313]: bool('Hello world!'), bool('')
Out[313]: (True, False)

In [314]: bool(0), bool(1)
Out[314]: (False, True)
```

Type casting

The `str`, `bool`, `int` and `float` types are also functions which can be used to cast values to those types:

```
In [315]: s = '3.14159'

In [316]: fval = float(s)      In [317]: type(fval)
Out[317]: float

In [318]: int(fval)           In [319]: bool(fval)           In [320]: bool(0)
Out[318]: 3                   Out[319]: True              Out[320]: False
```

None

`None` is the Python null value type. If a function does not explicitly return a value, it implicitly returns `None`.

```
In [321]: a = None           In [322]: a is None
Out[322]: True

In [323]: b = 5              In [324]: b is not None
Out[324]: True
```

`None` is also a common default value for optional function arguments:

```
def add_and_maybe_multiply(a, b, c=None):
    result = a + b

    if c is not None:
        result = result * c

    return result
```

While a technical point, it's worth bearing in mind that `None` is not a reserved keyword but rather a unique instance of `NoneType`.

Dates and times

The built-in Python `datetime` module provides `datetime`, `date`, and `time` types. The `datetime` type as you may imagine combines the information stored in `date` and `time` and is the most commonly used:

```
In [325]: from datetime import datetime, date, time

In [326]: dt = datetime(2011, 10, 29, 20, 30, 21)
```

```
In [327]: dt.day      In [328]: dt.minute
Out[327]: 29         Out[328]: 30
```

Given a `datetime` instance, you can extract the equivalent `date` and `time` objects by calling methods on the `datetime` of the same name:

```
In [329]: dt.date()      In [330]: dt.time()
Out[329]: datetime.date(2011, 10, 29)  Out[330]: datetime.time(20, 30, 21)
```

The `strftime` method formats a `datetime` as a string:

```
In [331]: dt.strftime('%m/%d/%Y %H:%M')
Out[331]: '10/29/2011 20:30'
```

Strings can be converted (parsed) into `datetime` objects using the `strptime` function:

```
In [332]: datetime.strptime('20091031', '%Y%m%d')
Out[332]: datetime.datetime(2009, 10, 31, 0, 0)
```

See [Table 10-2](#) for a full list of format specifications.

When aggregating or otherwise grouping time series data, it will occasionally be useful to replace fields of a series of datetimes, for example replacing the minute and second fields with zero, producing a new object:

```
In [333]: dt.replace(minute=0, second=0)
Out[333]: datetime.datetime(2011, 10, 29, 20, 0)
```

The difference of two `datetime` objects produces a `datetime.timedelta` type:

```
In [334]: dt2 = datetime(2011, 11, 15, 22, 30)

In [335]: delta = dt2 - dt

In [336]: delta      In [337]: type(delta)
Out[336]: datetime.timedelta(17, 7179)  Out[337]: datetime.timedelta
```

Adding a `timedelta` to a `datetime` produces a new shifted `datetime`:

```
In [338]: dt
Out[338]: datetime.datetime(2011, 10, 29, 20, 30, 21)

In [339]: dt + delta
Out[339]: datetime.datetime(2011, 11, 15, 22, 30)
```

Control Flow

if, elif, and else

The `if` statement is one of the most well-known control flow statement types. It checks a condition which, if `True`, evaluates the code in the block that follows:

```
if x < 0:
    print 'It's negative'
```

An `if` statement can be optionally followed by one or more `elif` blocks and a catch-all `else` block if all of the conditions are `False`:

```
if x < 0:
    print 'It's negative'
elif x == 0:
    print 'Equal to zero'
elif 0 < x < 5:
    print 'Positive but smaller than 5'
else:
    print 'Positive and larger than or equal to 5'
```

If any of the conditions is `True`, no further `elif` or `else` blocks will be reached. With a compound condition using `and` or `or`, conditions are evaluated left-to-right and will short circuit:

```
In [340]: a = 5; b = 7

In [341]: c = 8; d = 4

In [342]: if a < b or c > d:
.....:     print 'Made it'
Made it
```

In this example, the comparison `c > d` never gets evaluated because the first comparison was `True`.

for loops

`for` loops are for iterating over a collection (like a list or tuple) or an iterator. The standard syntax for a `for` loop is:

```
for value in collection:
    # do something with value
```

A `for` loop can be advanced to the next iteration, skipping the remainder of the block, using the `continue` keyword. Consider this code which sums up integers in a list and skips `None` values:

```
sequence = [1, 2, None, 4, None, 5]
total = 0
for value in sequence:
    if value is None:
        continue
    total += value
```

A `for` loop can be exited altogether using the `break` keyword. This code sums elements of the list until a 5 is reached:

```
sequence = [1, 2, 0, 4, 6, 5, 2, 1]
total_until_5 = 0
for value in sequence:
    if value == 5:
        break
    total_until_5 += value
```

As we will see in more detail, if the elements in the collection or iterator are sequences (tuples or lists, say), they can be conveniently *unpacked* into variables in the `for` loop statement:

```
for a, b, c in iterator:
    # do something
```

while loops

A `while` loop specifies a condition and a block of code that is to be executed until the condition evaluates to `False` or the loop is explicitly ended with `break`:

```
x = 256
total = 0
while x > 0:
    if total > 500:
        break
    total += x
    x = x // 2
```

pass

`pass` is the “no-op” statement in Python. It can be used in blocks where no action is to be taken; it is only required because Python uses whitespace to delimit blocks:

```
if x < 0:
    print 'negative!'
elif x == 0:
    # TODO: put something smart here
    pass
else:
    print 'positive!'
```

It's common to use `pass` as a place-holder in code while working on a new piece of functionality:

```
def f(x, y, z):
    # TODO: implement this function!
    pass
```

Exception handling

Handling Python errors or *exceptions* gracefully is an important part of building robust programs. In data analysis applications, many functions only work on certain kinds of input. As an example, Python's `float` function is capable of casting a string to a floating point number, but fails with `ValueError` on improper inputs:

```
In [343]: float('1.2345')
Out[343]: 1.2345

In [344]: float('something')
-----
ValueError                                Traceback (most recent call last)
<ipython-input-344-439904410854> in <module>()
```

```
----> 1 float('something')
ValueError: could not convert string to float: something
```

Suppose we wanted a version of `float` that fails gracefully, returning the input argument. We can do this by writing a function that encloses the call to `float` in a `try/except` block:

```
def attempt_float(x):
    try:
        return float(x)
    except:
        return x
```

The code in the `except` part of the block will only be executed if `float(x)` raises an exception:

```
In [346]: attempt_float('1.2345')
Out[346]: 1.2345

In [347]: attempt_float('something')
Out[347]: 'something'
```

You might notice that `float` can raise exceptions other than `ValueError`:

```
In [348]: float((1, 2))
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-348-842079ebb635> in <module>()
----> 1 float((1, 2))
TypeError: float() argument must be a string or a number
```

You might want to only suppress `ValueError`, since a `TypeError` (the input was not a string or numeric value) might indicate a legitimate bug in your program. To do that, write the exception type after `except`:

```
def attempt_float(x):
    try:
        return float(x)
    except ValueError:
        return x
```

We have then:

```
In [350]: attempt_float((1, 2))
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-350-9bdfd730cead> in <module>()
----> 1 attempt_float((1, 2))
<ipython-input-349-3e06b8379b6b> in attempt_float(x)
      1 def attempt_float(x):
      2     try:
----> 3         return float(x)
      4     except ValueError:
      5         return x
TypeError: float() argument must be a string or a number
```


You can catch multiple exception types by writing a tuple of exception types instead (the parentheses are required):

```
def attempt_float(x):
    try:
        return float(x)
    except (TypeError, ValueError):
        return x
```

In some cases, you may not want to suppress an exception, but you want some code to be executed regardless of whether the code in the `try` block succeeds or not. To do this, use `finally`:

```
f = open(path, 'w')

try:
    write_to_file(f)
finally:
    f.close()
```

Here, the file handle `f` will *always* get closed. Similarly, you can have code that executes only if the `try` block succeeds using `else`:

```
f = open(path, 'w')

try:
    write_to_file(f)
except:
    print 'Failed'
else:
    print 'Succeeded'
finally:
    f.close()
```

range and xrange

The `range` function produces a list of evenly-spaced integers:

```
In [352]: range(10)
Out[352]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Both a start, end, and step can be given:

```
In [353]: range(0, 20, 2)
Out[353]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

As you can see, `range` produces integers up to but not including the endpoint. A common use of `range` is for iterating through sequences by index:

```
seq = [1, 2, 3, 4]
for i in range(len(seq)):
    val = seq[i]
```

For very long ranges, it's recommended to use `xrange`, which takes the same arguments as `range` but returns an iterator that generates integers one by one rather than generating

all of them up-front and storing them in a (potentially very large) list. This snippet sums all numbers from 0 to 9999 that are multiples of 3 or 5:

```
sum = 0
for i in xrange(10000):
    # % is the modulo operator
    if i % 3 == 0 or i % 5 == 0:
        sum += i
```



In Python 3, `range` always returns an iterator, and thus it is not necessary to use the `xrange` function

Ternary Expressions

A *ternary expression* in Python allows you combine an `if-else` block which produces a value into a single line or expression. The syntax for this in Python is

```
value = true-expr if condition else
       false-expr
```

Here, *true-expr* and *false-expr* can be any Python expressions. It has the identical effect as the more verbose

```
if condition:
    value = true-expr
else:
    value = false-expr
```

This is a more concrete example:

```
In [354]: x = 5

In [355]: 'Non-negative' if x >= 0 else 'Negative'
Out[355]: 'Non-negative'
```

As with `if-else` blocks, only one of the expressions will be evaluated. While it may be tempting to always use ternary expressions to condense your code, realize that you may sacrifice readability if the condition as well and the true and false expressions are very complex.

Data Structures and Sequences

Python's data structures are simple, but powerful. Mastering their use is a critical part of becoming a proficient Python programmer.

Tuple

A tuple is a one-dimensional, fixed-length, *immutable* sequence of Python objects. The easiest way to create one is with a comma-separated sequence of values:

```
In [356]: tup = 4, 5, 6
```

```
In [357]: tup
Out[357]: (4, 5, 6)
```

When defining tuples in more complicated expressions, it's often necessary to enclose the values in parentheses, as in this example of creating a tuple of tuples:

```
In [358]: nested_tup = (4, 5, 6), (7, 8)
```

```
In [359]: nested_tup
Out[359]: ((4, 5, 6), (7, 8))
```

Any sequence or iterator can be converted to a tuple by invoking `tuple`:

```
In [360]: tuple([4, 0, 2])
Out[360]: (4, 0, 2)
```

```
In [361]: tup = tuple('string')
```

```
In [362]: tup
Out[362]: ('s', 't', 'r', 'i', 'n', 'g')
```

Elements can be accessed with square brackets `[]` as with most other sequence types. Like C, C++, Java, and many other languages, sequences are 0-indexed in Python:

```
In [363]: tup[0]
Out[363]: 's'
```

While the objects stored in a tuple may be mutable themselves, once created it's not possible to modify which object is stored in each slot:

```
In [364]: tup = tuple(['foo', [1, 2], True])
```

```
In [365]: tup[2] = False
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-365-c7308343b841> in <module>()
----> 1 tup[2] = False
TypeError: 'tuple' object does not support item assignment
```

```
# however
```

```
In [366]: tup[1].append(3)
```

```
In [367]: tup
Out[367]: ('foo', [1, 2, 3], True)
```

Tuples can be concatenated using the `+` operator to produce longer tuples:

```
In [368]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[368]: (4, None, 'foo', 6, 0, 'bar')
```

Multiplying a tuple by an integer, as with lists, has the effect of concatenating together that many copies of the tuple.

```
In [369]: ('foo', 'bar') * 4
Out[369]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

Note that the objects themselves are not copied, only the references to them.

Unpacking tuples

If you try to *assign* to a tuple-like expression of variables, Python will attempt to *unpack* the value on the right-hand side of the equals sign:

```
In [370]: tup = (4, 5, 6)

In [371]: a, b, c = tup

In [372]: b
Out[372]: 5
```

Even sequences with nested tuples can be unpacked:

```
In [373]: tup = 4, 5, (6, 7)

In [374]: a, b, (c, d) = tup

In [375]: d
Out[375]: 7
```

Using this functionality it's easy to swap variable names, a task which in many languages might look like:

```
tmp = a
a = b
b = tmp

b, a = a, b
```

One of the most common uses of variable unpacking when iterating over sequences of tuples or lists:

```
seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
for a, b, c in seq:
    pass
```

Another common use is for returning multiple values from a function. More on this later.

Tuple methods

Since the size and contents of a tuple cannot be modified, it is very light on instance methods. One particularly useful one (also available on lists) is `count`, which counts the number of occurrences of a value:

```
In [376]: a = (1, 2, 2, 2, 3, 4, 2)
```

```
In [377]: a.count(2)
Out[377]: 4
```

List

In contrast with tuples, lists are variable-length and their contents can be modified. They can be defined using square brackets `[]` or using the `list` type function:

```
In [378]: a_list = [2, 3, 7, None]

In [379]: tup = ('foo', 'bar', 'baz')

In [380]: b_list = list(tup)      In [381]: b_list
Out[381]: ['foo', 'bar', 'baz']

In [382]: b_list[1] = 'peekaboo' In [383]: b_list
Out[383]: ['foo', 'peekaboo', 'baz']
```

Lists and tuples are semantically similar as one-dimensional sequences of objects and thus can be used interchangeably in many functions.

Adding and removing elements

Elements can be appended to the end of the list with the `append` method:

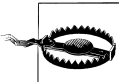
```
In [384]: b_list.append('dwarf')

In [385]: b_list
Out[385]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

Using `insert` you can insert an element at a specific location in the list:

```
In [386]: b_list.insert(1, 'red')

In [387]: b_list
Out[387]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```



`insert` is computationally expensive compared with `append` as references to subsequent elements have to be shifted internally to make room for the new element.

The inverse operation to `insert` is `pop`, which removes and returns an element at a particular index:

```
In [388]: b_list.pop(2)
Out[388]: 'peekaboo'

In [389]: b_list
Out[389]: ['foo', 'red', 'baz', 'dwarf']
```

Elements can be removed by value using `remove`, which locates the first such value and removes it from the list:

```
In [390]: b_list.append('foo')

In [391]: b_list.remove('foo')

In [392]: b_list
Out[392]: ['red', 'baz', 'dwarf', 'foo']
```

If performance is not a concern, by using `append` and `remove`, a Python list can be used as a perfectly suitable “multi-set” data structure.

You can check if a list contains a value using the `in` keyword:

```
In [393]: 'dwarf' in b_list
Out[393]: True
```

Note that checking whether a list contains a value is a lot slower than dicts and sets as Python makes a linear scan across the values of the list, whereas the others (based on hash tables) can make the check in constant time.

Concatenating and combining lists

Similar to tuples, adding two lists together with `+` concatenates them:

```
In [394]: [4, None, 'foo'] + [7, 8, (2, 3)]
Out[394]: [4, None, 'foo', 7, 8, (2, 3)]
```

If you have a list already defined, you can append multiple elements to it using the `extend` method:

```
In [395]: x = [4, None, 'foo']

In [396]: x.extend([7, 8, (2, 3)])

In [397]: x
Out[397]: [4, None, 'foo', 7, 8, (2, 3)]
```

Note that list concatenation is a comparatively expensive operation since a new list must be created and the objects copied over. Using `extend` to append elements to an existing list, especially if you are building up a large list, is usually preferable. Thus,

```
everything = []
for chunk in list_of_lists:
    everything.extend(chunk)
```

is faster than the concatenative alternative

```
everything = []
for chunk in list_of_lists:
    everything = everything + chunk
```

Sorting

A list can be sorted in-place (without creating a new object) by calling its `sort` function:

```
In [398]: a = [7, 2, 5, 1, 3]
```

```
In [399]: a.sort()

In [400]: a
Out[400]: [1, 2, 3, 5, 7]
```

`sort` has a few options that will occasionally come in handy. One is the ability to pass a secondary *sort key*, i.e. a function that produces a value to use to sort the objects. For example, we could sort a collection of strings by their lengths:

```
In [401]: b = ['saw', 'small', 'He', 'foxes', 'six']

In [402]: b.sort(key=len)

In [403]: b
Out[403]: ['He', 'saw', 'six', 'small', 'foxes']
```

Binary search and maintaining a sorted list

The built-in `bisect` module implements binary-search and insertion into a sorted list. `bisect.bisect` finds the location where an element should be inserted to keep it sorted, while `bisect.insort` actually inserts the element into that location:

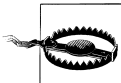
```
In [404]: import bisect

In [405]: c = [1, 2, 2, 2, 3, 4, 7]

In [406]: bisect.bisect(c, 2)      In [407]: bisect.bisect(c, 5)
Out[406]: 4                      Out[407]: 6

In [408]: bisect.insort(c, 6)

In [409]: c
Out[409]: [1, 2, 2, 2, 3, 4, 6, 7]
```



The `bisect` module functions do not check whether the list is sorted as doing so would be computationally expensive. Thus, using them with an unsorted list will succeed without error but may lead to incorrect results.

Slicing

You can select sections of list-like types (arrays, tuples, NumPy arrays) by using slice notation, which in its basic form consists of `start:stop` passed to the indexing operator `[]`:

```
In [410]: seq = [7, 2, 3, 7, 5, 6, 0, 1]

In [411]: seq[1:5]
Out[411]: [2, 3, 7, 5]
```

Slices can also be assigned to with a sequence:

```
In [412]: seq[3:4] = [6, 3]
```

```
In [413]: seq
Out[413]: [7, 2, 3, 6, 3, 5, 6, 0, 1]
```

While element at the **start** index is included, the **stop** index is not included, so that the number of elements in the result is **stop - start**.

Either the **start** or **stop** can be omitted in which case they default to the start of the sequence and the end of the sequence, respectively:

```
In [414]: seq[:5]          In [415]: seq[3:]
Out[414]: [7, 2, 3, 6, 3]  Out[415]: [6, 3, 5, 6, 0, 1]
```

Negative indices slice the sequence relative to the end:

```
In [416]: seq[-4:]         In [417]: seq[-6:-2]
Out[416]: [5, 6, 0, 1]    Out[417]: [6, 3, 5, 6]
```

Slicing semantics takes a bit of getting used to, especially if you're coming from R or MATLAB. See [Figure A-2](#) for a helpful illustrating of slicing with positive and negative integers.

A **step** can also be used after a second colon to, say, take every other element:

```
In [418]: seq[::2]
Out[418]: [7, 3, 3, 6, 1]
```

A clever use of this is to pass **-1** which has the useful effect of reversing a list or tuple:

```
In [419]: seq[::-1]
Out[419]: [1, 0, 6, 5, 3, 6, 3, 2, 7]
```

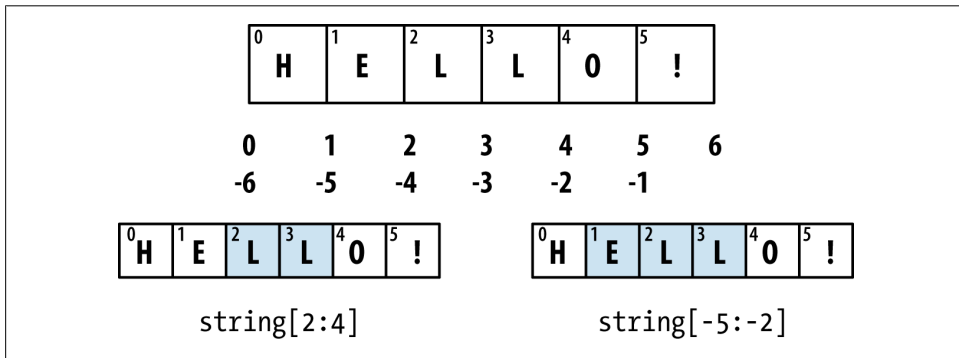


Figure A-2. Illustration of Python slicing conventions

Built-in Sequence Functions

Python has a handful of useful sequence functions that you should familiarize yourself with and use at any opportunity.

enumerate

It's common when iterating over a sequence to want to keep track of the index of the current item. A do-it-yourself approach would look like:

```
i = 0
for value in collection:
    # do something with value
    i += 1
```

Since this is so common, Python has a built-in function `enumerate` which returns a sequence of `(i, value)` tuples:

```
for i, value in enumerate(collection):
    # do something with value
```

When indexing data, a useful pattern that uses `enumerate` is computing a `dict` mapping the values of a sequence (which are assumed to be unique) to their locations in the sequence:

```
In [420]: some_list = ['foo', 'bar', 'baz']

In [421]: mapping = dict((v, i) for i, v in enumerate(some_list))

In [422]: mapping
Out[422]: {'bar': 1, 'baz': 2, 'foo': 0}
```

sorted

The `sorted` function returns a new sorted list from the elements of any sequence:

```
In [423]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[423]: [0, 1, 2, 2, 3, 6, 7]

In [424]: sorted('horse race')
Out[424]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

A common pattern for getting a sorted list of the unique elements in a sequence is to combine `sorted` with `set`:

```
In [425]: sorted(set('this is just some string'))
Out[425]: [' ', 'e', 'g', 'h', 'i', 'j', 'm', 'n', 'o', 'r', 's', 't', 'u']
```

zip

`zip` “pairs” up the elements of a number of lists, tuples, or other sequences, to create a list of tuples:

```
In [426]: seq1 = ['foo', 'bar', 'baz']

In [427]: seq2 = ['one', 'two', 'three']

In [428]: zip(seq1, seq2)
Out[428]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

`zip` can take an arbitrary number of sequences, and the number of elements it produces is determined by the *shortest* sequence:

```
In [429]: seq3 = [False, True]

In [430]: zip(seq1, seq2, seq3)
Out[430]: [('foo', 'one', False), ('bar', 'two', True)]
```

A very common use of `zip` is for simultaneously iterating over multiple sequences, possibly also combined with `enumerate`:

```
In [431]: for i, (a, b) in enumerate(zip(seq1, seq2)):
.....:     print('%d: %s, %s' % (i, a, b))
.....:
0: foo, one
1: bar, two
2: baz, three
```

Given a “zipped” sequence, `zip` can be applied in a clever way to “unzip” the sequence. Another way to think about this is converting a list of *rows* into a list of *columns*. The syntax, which looks a bit magical, is:

```
In [432]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),
.....:                ('Schilling', 'Curt')]

In [433]: first_names, last_names = zip(*pitchers)

In [434]: first_names
Out[434]: ('Nolan', 'Roger', 'Schilling')

In [435]: last_names
Out[435]: ('Ryan', 'Clemens', 'Curt')
```

We’ll look in more detail at the use of `*` in a function call. It is equivalent to the following:

```
zip(seq[0], seq[1], ..., seq[len(seq) - 1])
```

reversed

`reversed` iterates over the elements of a sequence in reverse order:

```
In [436]: list(reversed(range(10)))
Out[436]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Dict

`dict` is likely the most important built-in Python data structure. A more common name for it is *hash map* or *associative array*. It is a flexibly-sized collection of *key-value* pairs, where *key* and *value* are Python objects. One way to create one is by using curly braces `{}` and using colons to separate keys and values:

```
In [437]: empty_dict = {}

In [438]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}
```

```
In [439]: d1
Out[439]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

Elements can be accessed and inserted or set using the same syntax as accessing elements of a list or tuple:

```
In [440]: d1[7] = 'an integer'

In [441]: d1
Out[441]: {7: 'an integer', 'a': 'some value', 'b': [1, 2, 3, 4]}

In [442]: d1['b']
Out[442]: [1, 2, 3, 4]
```

You can check if a dict contains a key using the same syntax as with checking whether a list or tuple contains a value:

```
In [443]: 'b' in d1
Out[443]: True
```

Values can be deleted either using the `del` keyword or the `pop` method (which simultaneously returns the value and deletes the key):

```
In [444]: d1[5] = 'some value'

In [445]: d1['dummy'] = 'another value'

In [446]: del d1[5]

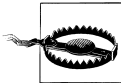
In [447]: ret = d1.pop('dummy')
Out[447]: 'another value'

In [448]: ret
Out[448]: 'another value'
```

The `keys` and `values` method give you lists of the keys and values, respectively. While the key-value pairs are not in any particular order, these functions output the keys and values in the same order:

```
In [449]: d1.keys()
Out[449]: ['a', 'b', 7]

In [450]: d1.values()
Out[450]: ['some value', [1, 2, 3, 4], 'an integer']
```



If you're using Python 3, `dict.keys()` and `dict.values()` are iterators instead of lists.

One dict can be merged into another using the `update` method:

```
In [451]: d1.update({'b': 'foo', 'c': 12})

In [452]: d1
Out[452]: {7: 'an integer', 'a': 'some value', 'b': 'foo', 'c': 12}
```

Creating dicts from sequences

It's common to occasionally end up with two sequences that you want to pair up element-wise in a dict. As a first cut, you might write code like this:

```
mapping = {}
for key, value in zip(key_list, value_list):
    mapping[key] = value
```

Since a dict is essentially a collection of 2-tuples, it should be no shock that the `dict` type function accepts a list of 2-tuples:

```
In [453]: mapping = dict(zip(range(5), reversed(range(5))))
```

```
In [454]: mapping
Out[454]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

In a later section we'll talk about *dict comprehensions*, another elegant way to construct dicts.

Default values

It's very common to have logic like:

```
if key in some_dict:
    value = some_dict[key]
else:
    value = default_value
```

Thus, the dict methods `get` and `pop` can take a default value to be returned, so that the above if-else block can be written simply as:

```
value = some_dict.get(key, default_value)
```

`get` by default will return `None` if the key is not present, while `pop` will raise an exception. With *setting* values, a common case is for the values in a dict to be other collections, like lists. For example, you could imagine categorizing a list of words by their first letters as a dict of lists:

```
In [455]: words = ['apple', 'bat', 'bar', 'atom', 'book']
```

```
In [456]: by_letter = {}
```

```
In [457]: for word in words:
.....:     letter = word[0]
.....:     if letter not in by_letter:
.....:         by_letter[letter] = [word]
.....:     else:
.....:         by_letter[letter].append(word)
.....:
```

```
In [458]: by_letter
Out[458]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

The `setdefault` dict method is for precisely this purpose. The if-else block above can be rewritten as:

```
by_letter.setdefault(letter, []).append(word)
```

The built-in `collections` module has a useful class, `defaultdict`, which makes this even easier. One is created by passing a type or function for generating the default value for each slot in the dict:

```
from collections import defaultdict
by_letter = defaultdict(list)
for word in words:
    by_letter[word[0]].append(word)
```

The initializer to `defaultdict` only needs to be a callable object (e.g. any function), not necessarily a type. Thus, if you wanted the default value to be 4 you could pass a function returning 4

```
counts = defaultdict(lambda: 4)
```

Valid dict key types

While the values of a dict can be any Python object, the keys have to be immutable objects like scalar types (int, float, string) or tuples (all the objects in the tuple need to be immutable, too). The technical term here is *hashability*. You can check whether an object is hashable (can be used as a key in a dict) with the `hash` function:

```
In [459]: hash('string')
Out[459]: -9167918882415130555

In [460]: hash((1, 2, (2, 3)))
Out[460]: 1097636502276347782

In [461]: hash((1, 2, [2, 3])) # fails because lists are mutable
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-461-800cd14ba8be> in <module>()
----> 1 hash((1, 2, [2, 3])) # fails because lists are mutable
TypeError: unhashable type: 'list'
```

To use a list as a key, an easy fix is to convert it to a tuple:

```
In [462]: d = {}

In [463]: d[tuple([1, 2, 3])] = 5

In [464]: d
Out[464]: {(1, 2, 3): 5}
```

Set

A set is an unordered collection of unique elements. You can think of them like dicts, but keys only, no values. A set can be created in two ways: via the `set` function or using a *set literal* with curly braces:

```
In [465]: set([2, 2, 2, 1, 3, 3])
Out[465]: set([1, 2, 3])
```

```
In [466]: {2, 2, 2, 1, 3, 3}
Out[466]: set([1, 2, 3])
```

Sets support mathematical *set operations* like union, intersection, difference, and symmetric difference. See [Table A-3](#) for a list of commonly used set methods.

```
In [467]: a = {1, 2, 3, 4, 5}
```

```
In [468]: b = {3, 4, 5, 6, 7, 8}
```

```
In [469]: a | b # union (or)
Out[469]: set([1, 2, 3, 4, 5, 6, 7, 8])
```

```
In [470]: a & b # intersection (and)
Out[470]: set([3, 4, 5])
```

```
In [471]: a - b # difference
Out[471]: set([1, 2])
```

```
In [472]: a ^ b # symmetric difference (xor)
Out[472]: set([1, 2, 6, 7, 8])
```

You can also check if a set is a subset of (is contained in) or a superset of (contains all elements of) another set:

```
In [473]: a_set = {1, 2, 3, 4, 5}
```

```
In [474]: {1, 2, 3}.issubset(a_set)
Out[474]: True
```

```
In [475]: a_set.issuperset({1, 2, 3})
Out[475]: True
```

As you might guess, sets are equal if their contents are equal:

```
In [476]: {1, 2, 3} == {3, 2, 1}
Out[476]: True
```

Table A-3. Python Set Operations

Function	Alternate Syntax	Description
<code>a.add(x)</code>	N/A	Add element <code>x</code> to the set <code>a</code>
<code>a.remove(x)</code>	N/A	Remove element <code>x</code> from the set <code>a</code>
<code>a.union(b)</code>	<code>a b</code>	All of the unique elements in <code>a</code> and <code>b</code> .
<code>a.intersection(b)</code>	<code>a & b</code>	All of the elements in <i>both</i> <code>a</code> and <code>b</code> .
<code>a.difference(b)</code>	<code>a - b</code>	The elements in <code>a</code> that are not in <code>b</code> .
<code>a.symmetric_difference(b)</code>	<code>a ^ b</code>	All of the elements in <code>a</code> or <code>b</code> but <i>not both</i> .
<code>a.issubset(b)</code>	N/A	True if the elements of <code>a</code> are all contained in <code>b</code> .
<code>a.issuperset(b)</code>	N/A	True if the elements of <code>b</code> are all contained in <code>a</code> .
<code>a.isdisjoint(b)</code>	N/A	True if <code>a</code> and <code>b</code> have no elements in common.

List, Set, and Dict Comprehensions

List comprehensions are one of the most-loved Python language features. They allow you to concisely form a new list by filtering the elements of a collection and transforming the elements passing the filter in one concise expression. They take the basic form:

```
[expr for val in collection if condition]
```

This is equivalent to the following `for` loop:

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

The filter condition can be omitted, leaving only the expression. For example, given a list of strings, we could filter out strings with length 2 or less and also convert them to uppercase like this:

```
In [477]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
```

```
In [478]: [x.upper() for x in strings if len(x) > 2]
Out[478]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Set and dict comprehensions are a natural extension, producing sets and dicts in a idiomatically similar way instead of lists. A dict comprehension looks like this:

```
dict_comp = {key-expr : value-expr for value in collection
              if condition}
```

A set comprehension looks like the equivalent list comprehension except with curly braces instead of square brackets:

```
set_comp = {expr for value in collection if condition}
```

Like list comprehensions, set and dict comprehensions are just syntactic sugar, but they similarly can make code both easier to write and read. Consider the list of strings above. Suppose we wanted a set containing just the lengths of the strings contained in the collection; this could be easily computed using a set comprehension:

```
In [479]: unique_lengths = {len(x) for x in strings}
```

```
In [480]: unique_lengths
Out[480]: set([1, 2, 3, 4, 6])
```

As a simple dict comprehension example, we could create a lookup map of these strings to their locations in the list:

```
In [481]: loc_mapping = {val : index for index, val in enumerate(strings)}
```

```
In [482]: loc_mapping
Out[482]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

Note that this dict could be equivalently constructed by:

```
loc_mapping = dict((val, idx) for idx, val in enumerate(strings))
```

The dict comprehension version is shorter and cleaner in my opinion.



Dict and set comprehensions were added to Python fairly recently in Python 2.7 and Python 3.1+.

Nested list comprehensions

Suppose we have a list of lists containing some boy and girl names:

```
In [483]: all_data = [['Tom', 'Billy', 'Jefferson', 'Andrew', 'Wesley', 'Steven', 'Joe'],
.....:               ['Susie', 'Casey', 'Jill', 'Ana', 'Eva', 'Jennifer', 'Stephanie']]
```

You might have gotten these names from a couple of files and decided to keep the boy and girl names separate. Now, suppose we wanted to get a single list containing all names with two or more e's in them. We could certainly do this with a simple `for` loop:

```
names_of_interest = []
for names in all_data:
    enough_es = [name for name in names if name.count('e') >= 2]
    names_of_interest.extend(enough_es)
```

You can actually wrap this whole operation up in a single *nested list comprehension*, which will look like:

```
In [484]: result = [name for names in all_data for name in names
.....:               if name.count('e') >= 2]
```

```
In [485]: result
Out[485]: ['Jefferson', 'Wesley', 'Steven', 'Jennifer', 'Stephanie']
```

At first, nested list comprehensions are a bit hard to wrap your head around. The `for` parts of the list comprehension are arranged according to the order of nesting, and any filter condition is put at the end as before. Here is another example where we “flatten” a list of tuples of integers into a simple list of integers:

```
In [486]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

```
In [487]: flattened = [x for tup in some_tuples for x in tup]
```

```
In [488]: flattened
Out[488]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Keep in mind that the order of the `for` expressions would be the same if you wrote a nested `for` loop instead of a list comprehension:

```
flattened = []

for tup in some_tuples:
    for x in tup:
        flattened.append(x)
```


You can have arbitrarily many levels of nesting, though if you have more than two or three levels of nesting you should probably start to question your data structure design. It's important to distinguish the above syntax from a list comprehension inside a list comprehension, which is also perfectly valid:

```
In [229]: [[x for x in tup] for tup in some_tuples]
```

Functions

Functions are the primary and most important method of code organization and reuse in Python. There may not be such a thing as having too many functions. In fact, I would argue that most programmers doing data analysis don't write enough functions! As you have likely inferred from prior examples, functions are declared using the `def` keyword and returned from using the `return` keyword:

```
def my_function(x, y, z=1.5):
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```

There is no issue with having multiple `return` statements. If the end of a function is reached without encountering a `return` statement, `None` is returned.

Each function can have some number of *positional* arguments and some number of *keyword* arguments. Keyword arguments are most commonly used to specify default values or optional arguments. In the above function, `x` and `y` are positional arguments while `z` is a keyword argument. This means that it can be called in either of these equivalent ways:

```
my_function(5, 6, z=0.7)
my_function(3.14, 7, 3.5)
```

The main restriction on function arguments is that the keyword arguments *must* follow the positional arguments (if any). You can specify keyword arguments in any order; this frees you from having to remember which order the function arguments were specified in and only what their names are.

Namespaces, Scope, and Local Functions

Functions can access variables in two different scopes: *global* and *local*. An alternate and more descriptive name describing a variable scope in Python is a *namespace*. Any variables that are assigned within a function by default are assigned to the local namespace. The local namespace is created when the function is called and immediately populated by the function's arguments. After the function is finished, the local namespace is destroyed (with some exceptions, see section on closures below). Consider the following function:

```
def func():
    a = []
    for i in range(5):
        a.append(i)
```

Upon calling `func()`, the empty list `a` is created, 5 elements are appended, then `a` is destroyed when the function exits. Suppose instead we had declared `a`

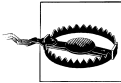
```
a = []
def func():
    for i in range(5):
        a.append(i)
```

Assigning global variables within a function is possible, but those variables must be declared as global using the `global` keyword:

```
In [489]: a = None

In [490]: def bind_a_variable():
.....:     global a
.....:     a = []
.....:     bind_a_variable()
.....:

In [491]: print a
[]
```



I generally discourage people from using the `global` keyword frequently. Typically global variables are used to store some kind of state in a system. If you find yourself using a lot of them, it's probably a sign that some object-oriented programming (using classes) is in order.

Functions can be declared anywhere, and there is no problem with having *local* functions that are dynamically created when a function is called:

```
def outer_function(x, y, z):
    def inner_function(a, b, c):
        pass
    pass
```

In the above code, the `inner_function` will not exist until `outer_function` is called. As soon as `outer_function` is done executing, the `inner_function` is destroyed.

Nested inner functions can access the local namespace of the enclosing function, but they cannot bind new variables in it. I'll talk a bit more about this in the section on closures.

In a strict sense, all functions are local to some scope, that scope may just be the module level scope.

Returning Multiple Values

When I first programmed in Python after having programmed in Java and C++, one of my favorite features was the ability to return multiple values from a function. Here's a simple example:

```
def f():
    a = 5
    b = 6
    c = 7
    return a, b, c

a, b, c = f()
```

In data analysis and other scientific applications, you will likely find yourself doing this very often as many functions may have multiple outputs, whether those are data structures or other auxiliary data computed inside the function. If you think about tuple packing and unpacking from earlier in this chapter, you may realize that what's happening here is that the function is actually just returning *one* object, namely a tuple, which is then being unpacked into the result variables. In the above example, we could have done instead:

```
return_value = f()
```

In this case, `return_value` would be, as you may guess, a 3-tuple with the three returned variables. A potentially attractive alternative to returning multiple values like above might be to return a dict instead:

```
def f():
    a = 5
    b = 6
    c = 7
    return {'a' : a, 'b' : b, 'c' : c}
```

Functions Are Objects

Since Python functions are objects, many constructs can be easily expressed that are difficult to do in other languages. Suppose we were doing some data cleaning and needed to apply a bunch of transformations to the following list of strings:

```
states = [' Alabama ', 'Georgia!', 'Georgia', 'georgia', 'Fl0rIda',
          'south carolina##', 'West virginia?']
```

Anyone who has ever worked with user-submitted survey data can expect messy results like these. Lots of things need to happen to make this list of strings uniform and ready for analysis: whitespace stripping, removing punctuation symbols, and proper capitalization. As a first pass, we might write some code like:

```
import re # Regular expression module

def clean_strings(strings):
    result = []
```

```

for value in strings:
    value = value.strip()
    value = re.sub('[!#?]', '', value) # remove punctuation
    value = value.title()
    result.append(value)
return result

```

The result looks like this:

```

In [15]: clean_strings(states)
Out[15]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']

```

An alternate approach that you may find useful is to make a list of the operations you want to apply to a particular set of strings:

```

def remove_punctuation(value):
    return re.sub('[!#?]', '', value)

clean_ops = [str.strip, remove_punctuation, str.title]

def clean_strings(strings, ops):
    result = []
    for value in strings:
        for function in ops:
            value = function(value)
        result.append(value)
    return result

```

Then we have

```

In [22]: clean_strings(states, clean_ops)
Out[22]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']

```

A more *functional* pattern like this enables you to easily modify how the strings are transformed at a very high level. The `clean_strings` function is also now more reusable!

You can naturally use functions as arguments to other functions like the built-in `map` function, which applies a function to a collection of some kind:

```

In [23]: map(remove_punctuation, states)
Out[23]:
[' Alabama ',
 'Georgia',

```

```
'Georgia',  
'georgia',  
'FLorida',  
'south carolina',  
'West virginia']
```

Anonymous (lambda) Functions

Python has support for so-called *anonymous* or *lambda* functions, which are really just simple functions consisting of a single statement, the result of which is the return value. They are defined using the `lambda` keyword, which has no meaning other than “we are declaring an anonymous function.”

```
def short_function(x):  
    return x * 2
```

```
equiv_anon = lambda x: x * 2
```

I usually refer to these as lambda functions in the rest of the book. They are especially convenient in data analysis because, as you’ll see, there are many cases where data transformation functions will take functions as arguments. It’s often less typing (and clearer) to pass a lambda function as opposed to writing a full-out function declaration or even assigning the lambda function to a local variable. For example, consider this silly example:

```
def apply_to_list(some_list, f):  
    return [f(x) for x in some_list]
```

```
ints = [4, 0, 1, 5, 6]  
apply_to_list(ints, lambda x: x * 2)
```

You could also have written `[x * 2 for x in ints]`, but here we were able to succinctly pass a custom operator to the `apply_to_list` function.

As another example, suppose you wanted to sort a collection of strings by the number of distinct letters in each string:

```
In [492]: strings = ['foo', 'card', 'bar', 'aaaa', 'abab']
```

Here we could pass a lambda function to the list’s `sort` method:

```
In [493]: strings.sort(key=lambda x: len(set(list(x))))
```

```
In [494]: strings  
Out[494]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```



One reason lambda functions are called anonymous functions is that the function object itself is never given a name attribute.

Closures: Functions that Return Functions

Closures are nothing to fear. They can actually be a very useful and powerful tool in the right circumstance! In a nutshell, a closure is any *dynamically-generated* function returned by another function. The key property is that the returned function has access to the variables in the local namespace where it was created. Here is a very simple example:

```
def make_closure(a):
    def closure():
        print('I know the secret: %d' % a)
    return closure

closure = make_closure(5)
```

The difference between a closure and a regular Python function is that the closure continues to have access to the namespace (the function) where it was created, even though that function is done executing. So in the above case, the returned closure will always print `I know the secret: 5` whenever you call it. While it's common to create closures whose internal state (in this example, only the value of `a`) is static, you can just as easily have a mutable object like a dict, set, or list that can be modified. For example, here's a function that returns a function that keeps track of arguments it has been called with:

```
def make_watcher():
    have_seen = {}

    def has_been_seen(x):
        if x in have_seen:
            return True
        else:
            have_seen[x] = True
            return False

    return has_been_seen
```

Using this on a sequence of integers I obtain:

```
In [496]: watcher = make_watcher()

In [497]: vals = [5, 6, 1, 5, 1, 6, 3, 5]

In [498]: [watcher(x) for x in vals]
Out[498]: [False, False, False, True, True, True, False, True]
```

However, one technical limitation to keep in mind is that while you can mutate any internal state objects (like adding key-value pairs to a dict), you cannot *bind* variables in the enclosing function scope. One way to work around this is to modify a dict or list rather than binding variables:

```
def make_counter():
    count = [0]
    def counter():
```

```

        # increment and return the current count
        count[0] += 1
        return count[0]
    return counter

```

```
counter = make_counter()
```

You might be wondering why this is useful. In practice, you can write very general functions with lots of options, then fabricate simpler, more specialized functions. Here's an example of creating a string formatting function:

```

def format_and_pad(template, space):
    def formatter(x):
        return (template % x).rjust(space)

    return formatter

```

You could then create a floating point formatter that always returns a length-15 string like so:

```
In [500]: fmt = format_and_pad('%.4f', 15)
```

```

In [501]: fmt(1.756)
Out[501]: '          1.7560'

```

If you learn more about object-oriented programming in Python, you might observe that these patterns also could be implemented (albeit more verbosely) using classes.

Extended Call Syntax with `*args`, `**kwargs`

The way that function arguments work under the hood in Python is actually very simple. When you write `func(a, b, c, d=some, e=value)`, the positional and keyword arguments are actually packed up into a tuple and dict, respectively. So the internal function receives a tuple `args` and dict `kwargs` and internally does the equivalent of:

```

a, b, c = args
d = kwargs.get('d', d_default_value)
e = kwargs.get('e', e_default_value)

```

This all happens nicely behind the scenes. Of course, it also does some error checking and allows you to specify some of the positional arguments as keywords also (even if they aren't keyword in the function declaration!).

```

def say_hello_then_call_f(f, *args, **kwargs):
    print 'args is', args
    print 'kwargs is', kwargs
    print("Hello! Now I'm going to call %s" % f)
    return f(*args, **kwargs)

def g(x, y, z=1):
    return (x + y) / z

```

Then if we call `g` with `say_hello_then_call_f` we get:

```
In [8]: say_hello_then_call_f(g, 1, 2, z=5.)
args is (1, 2)
kwargs is {'z': 5.0}
Hello! Now I'm going to call <function g at 0x2dd5cf8>
Out[8]: 0.6
```

Currying: Partial Argument Application

Currying is a fun computer science term which means deriving new functions from existing ones by *partial argument application*. For example, suppose we had a trivial function that adds two numbers together:

```
def add_numbers(x, y):
    return x + y
```

Using this function, we could derive a new function of one variable, `add_five`, that adds 5 to its argument:

```
add_five = lambda y: add_numbers(5, y)
```

The second argument to `add_numbers` is said to be *curried*. There's nothing very fancy here as we really only have defined a new function that calls an existing function. The built-in `functools` module can simplify this process using the `partial` function:

```
from functools import partial
add_five = partial(add_numbers, 5)
```

When discussing pandas and time series data, we'll use this technique to create specialized functions for transforming data series

```
# compute 60-day moving average of time series x
ma60 = lambda x: pandas.rolling_mean(x, 60)

# Take the 60-day moving average of all time series in data
data.apply(ma60)
```

Generators

Having a consistent way to iterate over sequences, like objects in a list or lines in a file, is an important Python feature. This is accomplished by means of the *iterator protocol*, a generic way to make objects iterable. For example, iterating over a dict yields the dict keys:

```
In [502]: some_dict = {'a': 1, 'b': 2, 'c': 3}

In [503]: for key in some_dict:
.....:     print key,
a c b
```

When you write `for key in some_dict`, the Python interpreter first attempts to create an iterator out of `some_dict`:

```
In [504]: dict_iterator = iter(some_dict)
```



```
In [505]: dict_iterator
Out[505]: <dictionary-keyiterator at 0x10a0a1578>
```

Any iterator is any object that will yield objects to the Python interpreter when used in a context like a `for` loop. Most methods expecting a list or list-like object will also accept any iterable object. This includes built-in methods such as `min`, `max`, and `sum`, and type constructors like `list` and `tuple`:

```
In [506]: list(dict_iterator)
Out[506]: ['a', 'c', 'b']
```

A *generator* is a simple way to construct a new iterable object. Whereas normal functions execute and return a single value, generators return a sequence of values lazily, pausing after each one until the next one is requested. To create a generator, use the `yield` keyword instead of `return` in a function:

```
def squares(n=10):
    print 'Generating squares from 1 to %d' % (n ** 2)
    for i in xrange(1, n + 1):
        yield i ** 2
```

When you actually call the generator, no code is immediately executed:

```
In [2]: gen = squares()

In [3]: gen
Out[3]: <generator object squares at 0x34c8280>
```

It is not until you request elements from the generator that it begins executing its code:

```
In [4]: for x in gen:
...:     print x,
...:
Generating squares from 0 to 100
1 4 9 16 25 36 49 64 81 100
```

As a less trivial example, suppose we wished to find all unique ways to make change for \$1 (100 cents) using an arbitrary set of coins. You can probably think of various ways to implement this and how to store the unique combinations as you come up with them. One way is to write a generator that yields lists of coins (represented as integers):

```
def make_change(amount, coins=[1, 5, 10, 25], hand=None):
    hand = [] if hand is None else hand
    if amount == 0:
        yield hand
    for coin in coins:
        # ensures we don't give too much change, and combinations are unique
        if coin > amount or (len(hand) > 0 and hand[-1] < coin):
            continue

        for result in make_change(amount - coin, coins=coins,
                                   hand=hand + [coin]):
            yield result
```

The details of the algorithm are not that important (can you think of a shorter way?). Then we can write:

```
In [508]: for way in make_change(100, coins=[10, 25, 50]):
.....:     print way
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
[25, 25, 10, 10, 10, 10, 10]
[25, 25, 25, 25]
[50, 10, 10, 10, 10, 10]
[50, 25, 25]
[50, 50]
```

```
In [509]: len(list(make_change(100)))
Out[509]: 242
```

Generator expressions

A simple way to make a generator is by using a *generator expression*. This is a generator analogue to list, dict and set comprehensions; to create one, enclose what would otherwise be a list comprehension with parenthesis instead of brackets:

```
In [510]: gen = (x ** 2 for x in xrange(100))
```

```
In [511]: gen
```

```
Out[511]: <generator object <genexpr> at 0x10a0a31e0>
```

This is completely equivalent to the following more verbose generator:

```
def _make_gen():
    for x in xrange(100):
        yield x ** 2
gen = _make_gen()
```

Generator expressions can be used inside any Python function that will accept a generator:

```
In [512]: sum(x ** 2 for x in xrange(100))
Out[512]: 328350
```

```
In [513]: dict((i, i ** 2) for i in xrange(5))
Out[513]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

itertools module

The standard library `itertools` module has a collection of generators for many common data algorithms. For example, `groupby` takes any sequence and a function; this groups consecutive elements in the sequence by return value of the function. Here's an example:

```
In [514]: import itertools
```

```
In [515]: first_letter = lambda x: x[0]
```

```
In [516]: names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']
```

```
In [517]: for letter, names in itertools.groupby(names, first_letter):
.....:     print letter, list(names) # names is a generator
A ['Alan', 'Adam']
```

```
W ['Wes', 'Will']
A ['Albert']
S ['Steven']
```

See [Table A-4](#) for a list of a few other `itertools` functions I’ve frequently found useful.

Table A-4. Some useful `itertools` functions

Function	Description
<code>imap(func, *iterables)</code>	Generator version of the built-in <code>map</code> ; applies <code>func</code> to each zipped tuple of the passed sequences.
<code>ifilter(func, iterable)</code>	Generator version of the built-in <code>filter</code> ; yields elements <code>x</code> for which <code>func(x)</code> is <code>True</code> .
<code>combinations(iterable, k)</code>	Generates a sequence of all possible <code>k</code> -tuples of elements in the iterable, ignoring order.
<code>permutations(iterable, k)</code>	Generates a sequence of all possible <code>k</code> -tuples of elements in the iterable, respecting order.
<code>groupby(iterable[, keyfunc])</code>	Generates (<code>key</code> , <code>sub-iterator</code>) for each unique key



In Python 3, several built-in functions (`zip`, `map`, `filter`) producing lists have been replaced by their generator versions found in `itertools` in Python 2.

Files and the operating system

Most of this book uses high-level tools like `pandas.read_csv` to read data files from disk into Python data structures. However, it’s important to understand the basics of how to work with files in Python. Fortunately, it’s very simple, which is part of why Python is so popular for text and file munging.

To open a file for reading or writing, use the built-in `open` function with either a relative or absolute file path:

```
In [518]: path = 'ch13/segismundo.txt'
```

```
In [519]: f = open(path)
```

By default, the file is opened in read-only mode `'r'`. We can then treat the file handle `f` like a list and iterate over the lines like so

```
for line in f:
    pass
```

The lines come out of the file with the end-of-line (EOL) markers intact, so you’ll often see code to get an EOL-free list of lines in a file like

```
In [520]: lines = [x.rstrip() for x in open(path)]
```

```
In [521]: lines
```

```

Out[521]:
['Sue\x03\xb1a el rico en su riqueza,',
 'que m\x03\xa1s cuidados le ofrece;',
 '',
 'sue\x03\xb1a el pobre que padece',
 'su miseria y su pobreza;',
 '',
 'sue\x03\xb1a el que a medrar empieza,',
 'sue\x03\xb1a el que afana y pretende,',
 'sue\x03\xb1a el que agravia y ofende,',
 '',
 'y en el mundo, en conclusi\x03\xb3n,',
 'todos sue\x03\xb1an lo que son,',
 'aunque ninguno lo entiende.',
 '']

```

If we had typed `f = open(path, 'w')`, a *new file* at `ch13/segismundo.txt` would have been created, overwriting any one in its place. See below for a list of all valid file read/write modes.

Table A-5. Python file modes

Mode	Description
r	Read-only mode
w	Write-only mode. Creates a new file (deleting any file with the same name)
a	Append to existing file (create it if it does not exist)
r+	Read and write
b	Add to mode for binary files, that is 'rb' or 'wb'
U	Use universal newline mode. Pass by itself 'U' or appended to one of the read modes like 'rU'

To write text to a file, you can use either the file's `write` or `writelines` methods. For example, we could create a version of `prof_mod.py` with no blank lines like so:

```

In [522]: with open('tmp.txt', 'w') as handle:
.....:     handle.writelines(x for x in open(path) if len(x) > 1)

In [523]: open('tmp.txt').readlines()
Out[523]:
['Sue\x03\xb1a el rico en su riqueza,\n',
 'que m\x03\xa1s cuidados le ofrece;\n',
 'sue\x03\xb1a el pobre que padece\n',
 'su miseria y su pobreza;\n',
 'sue\x03\xb1a el que a medrar empieza,\n',
 'sue\x03\xb1a el que afana y pretende,\n',
 'sue\x03\xb1a el que agravia y ofende,\n',
 'y en el mundo, en conclusi\x03\xb3n,\n',
 'todos sue\x03\xb1an lo que son,\n',
 'aunque ninguno lo entiende.\n']

```

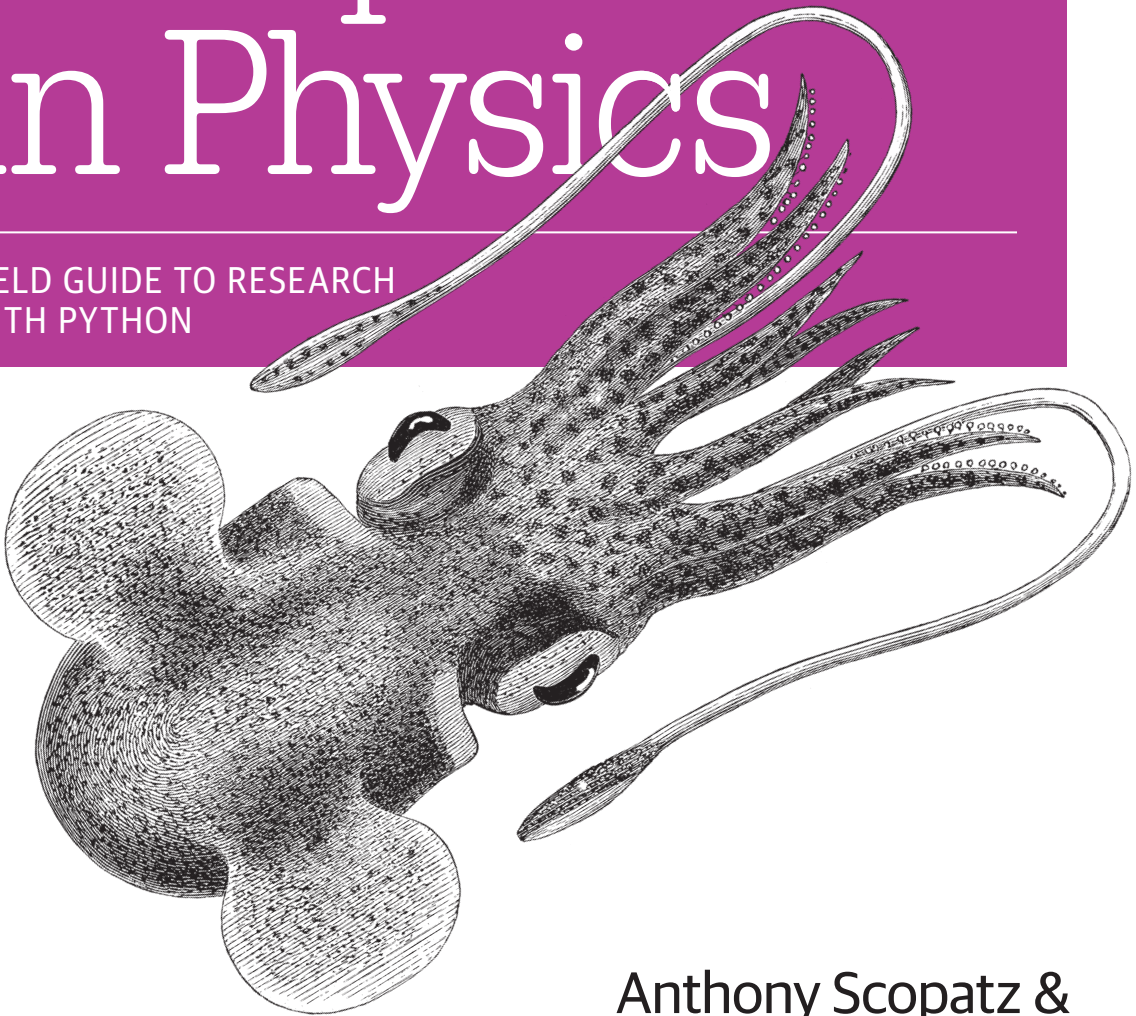
See [Table A-6](#) for many of the most commonly-used file methods.

Table A-6. Important Python file methods or attributes

Method	Description
<code>read([size])</code>	Return data from file as a string, with optional size argument indicating the number of bytes to read
<code>readlines([size])</code>	Return list of lines in the file, with optional size argument
<code>readlines([size])</code>	Return list of lines (as strings) in the file
<code>write(str)</code>	Write passed string to file.
<code>writelines(strings)</code>	Write passed sequence of strings to the file.
<code>close()</code>	Close the handle
<code>flush()</code>	Flush the internal I/O buffer to disk
<code>seek(pos)</code>	Move to indicated file position (integer).
<code>tell()</code>	Return current file position as integer.
<code>closed</code>	True if the file is closed.

Effective Computation in Physics

FIELD GUIDE TO RESEARCH
WITH PYTHON



Anthony Scopatz &
Kathryn D. Huff

Effective Computation in Physics

Anthony Scopatz and Kathryn D. Huff

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Introduction to the Command Line

The command line, or *shell*, provides a powerful, transparent interface between the user and the internals of a computer. At least on a Linux or Unix computer, the command line provides total access to the files and processes defining the state of the computer—including the files and processes of the operating system.

Also, many numerical tools for physics can only be installed and run through this interface. So, while this transparent interface could inspire the curiosity of a physicist all on its own, it is much more likely that you picked up this book because there is something you need to accomplish that only the command line will be capable of. While the command line may conjure images of *The Matrix*, do not let it intimidate you. Let's take the red pill.

Navigating the Shell

You can access the shell by opening a **terminal emulator** (“terminal” for short) on a Linux or Unix computer. On a Windows computer, the Git Bash program is equivalent. Launching the terminal opens an interactive shell program, which is where you will run your executable programs. The shell provides an interface, called the *command-line interface*, that can be used to run commands and navigate through the filesystem(s) to which your computer is connected. This command line is also sometimes called the *prompt*, and in this book it will be denoted with a dollar sign (\$) that points to where your cursor is ready to enter input. It should look something like **Figure 1-1**.

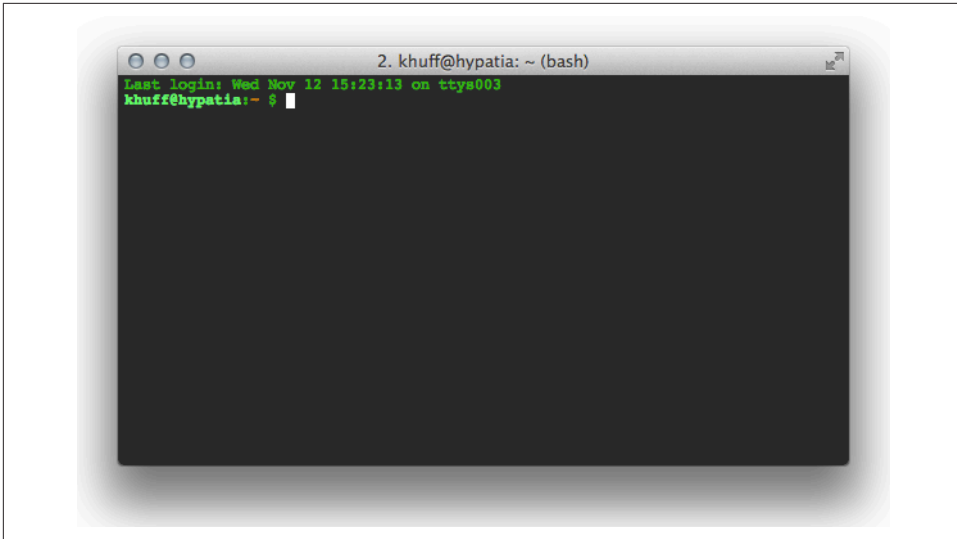


Figure 1-1. A terminal instance

This program is powerful and transparent, and provides total access to the files and processes on a computer. But what *is* the shell, exactly?

The Shell Is a Programming Language

The shell is a programming language that is run by the terminal. Like other programming languages, the shell:

- Can collect many operations into single entities
- Requires input
- Produces output
- Has variables and state
- Uses irritating syntax
- Uses special characters

Additionally, as with programming languages, there are more shells than you'll really care to learn. Among shells, **bash** is most widely used, so that is what we'll use in this discussion. The **csh**, **tcsh**, and **ksh** shell types are also popular. Features of various shells are listed in **Table 1-1**.

Table 1-1. Shell types

Shell	Name	Description
sh	Bourne shell	Popular, ubiquitous shell developed in 1977, still guaranteed on all Unixes
csh	C shell	Improves on sh
ksh	Korn shell	Backward-compatible with sh, but extends and borrows from other shells
bash	Bourne again shell	Free software replacement for sh, much evolved
tcsh	Tenex C shell	Updated and extended C shell



Exercise: Open a Terminal

1. Search your computer's programs to find one called Terminal. On a Windows computer, remember to use Git Bash as your bash terminal.
2. Open an instance of that program. You're in the shell!

The power of the shell resides in its transparency. By providing direct access to the entire filesystem, the shell can be used to accomplish nearly any task. Tasks such as finding files, manipulating them, installing libraries, and running programs begin with an understanding of paths and locations in the terminal.

Paths and pwd

The space where your files are—your file space—is made up of many nested directories (folders). In Unix parlance, the location of each directory (and each file inside them) is given by a “path.” These can be either **absolute paths** or **relative paths**.

Paths are *absolute* if they begin at the top of the filesystem directory tree. The very top of the filesystem directory tree is called the **root directory**. The path to the root directory is /. Therefore, absolute paths start with /.

In many UNIX and Linux systems, the root directory contains directories like *bin* and *lib*. The absolute paths to the *bin* and *lib* directories are then */bin* and */lib*, respectively. A diagram of an example directory tree, along with some notion of paths, can be seen in **Figure 1-2**.

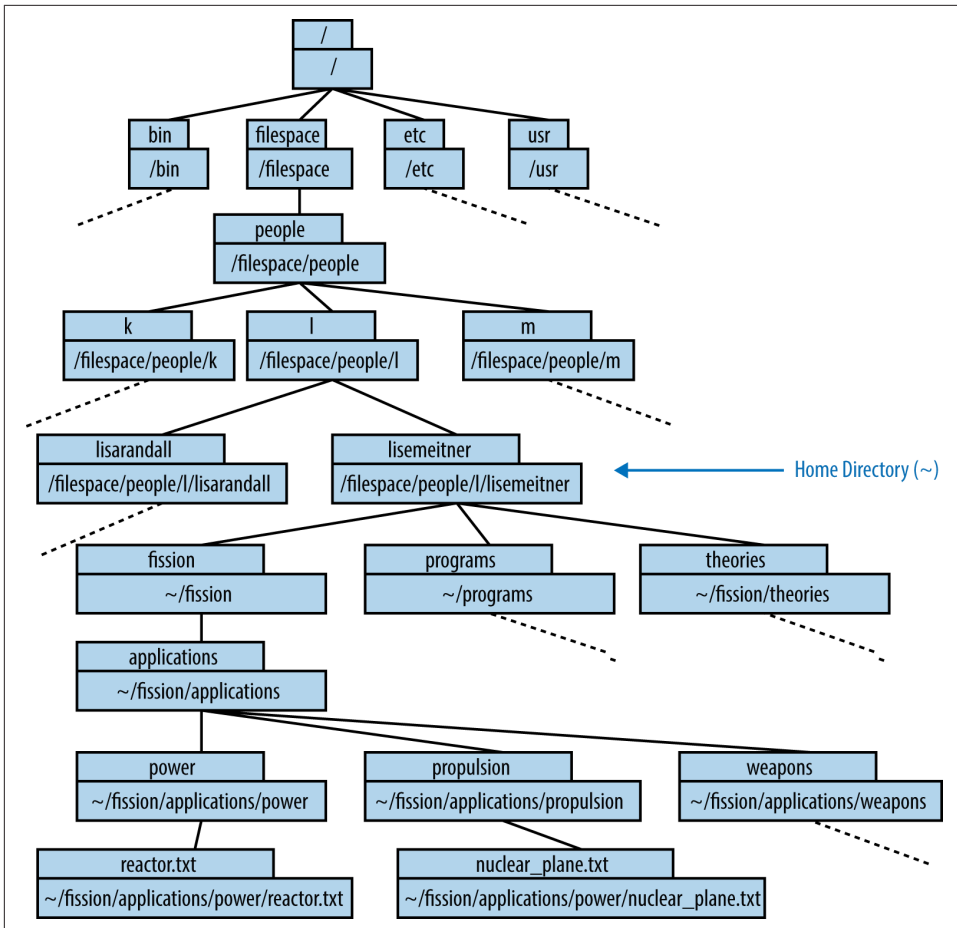


Figure 1-2. An example directory tree



The `/` syntax is used at the beginning of a path to indicate the top-level directory. It is also used to separate the names of directories in a path, as seen in [Figure 1-2](#).

Paths can, instead, be *relative* to your **current working directory**. The current working directory is denoted with one dot (`.`), while the directory immediately above it (its “parent”) is denoted with two dots (`..`). Relative paths therefore often start with a dot or two.

As we have learned, absolute paths describe a file space location relative to the root directory. Any path that describes a location relative to the current working directory

instead is a relative path. Bringing these together, note that you can always print out the full, absolute path of the directory you're currently working in with the command `pwd` (print working directory).

Bash was not available in the 1930s, when Lise Meitner was developing a theoretical framework for neutron-induced fission. However, had Bash been available, Prof. Meitner's research computer might have contained a set of directories holding files about her theory of fission as well as ideas about its application (see [Figure 1-2](#)). Let's take a look at how Lise would have navigated through this directory structure.



You can work along with Lise while you read this book. The directory tree she will be working with in this chapter is available in a repository on [GitHub](#). Read the instructions at that site to download the files.

When she is working, Lise enters commands at the command prompt. In the following example, we can see that the command prompt gives an abbreviated path name before the dollar sign (this is sometimes a greater-than sign or other symbol). That path is `~/fission`, because *fission* is the directory that Lise is currently working in:

```
~/fission $
```

When she types `pwd` at the command prompt, the shell returns (on the following line) the full path to her current working directory:

```
~/fission $ pwd
/filespace/people/l/lisemeitner/fission/
```

When we compare the absolute path and the abbreviated prompt, it seems that the prompt replaces all the directories up to and including *lisemeitner* with a single character, the tilde (`~`). In the next section, we'll see why.

Home Directory (~)

The shell starts your session from a special directory called your *home directory*. The tilde (`~`) character can be used as a shortcut to your home directory. Thus, when you log in, you probably see the command prompt telling you you're in your home directory:

```
~ $
```

These prompts are not universal. Sometimes, the prompt shows the username and the name of the computer as well:

```
<user>@<machine>:~ $
```

For Prof. Meitner, who held a research position at the prestigious Kaiser Wilhelm Institute, this might appear as:

```
meitner@kaiser-wilhelm-cluster:~ $
```

Returning to the previous example, let us compare:

```
~/fission
```

to:

```
/filespace/people/l/lisemeitner/fission
```

It seems that the tilde has entirely replaced the home directory path (*/filespace/people/l/lisemeitner*). Indeed, the tilde is an abbreviation for the home directory path—that is, the sequence of characters (also known as a **string**) beginning with the root directory (/). Because the path is defined relative to the absolute top of the directory tree, this:

```
~/fission
```

and this:

```
/filespace/people/l/lisemeitner/fission
```

are both absolute paths.



Exercise: Find Home

1. Open the Terminal.
2. Type `pwd` at the command prompt and press Enter to see the absolute path to your home directory.

Now that she knows where she is in the filesystem, curious Lise is interested in what she'll find there. To list the contents of a directory, she'll need the `ls` command.

Listing the Contents (`ls`)

The `ls` command allows the user to print out a list of all the files and subdirectories in a directory.



Exercise: List the Contents of a Directory

1. Open the Terminal.
2. Type `ls` at the command prompt and press Enter to see the contents of your home directory.

From the *fission* directory in Professor Meitner's home directory, `ls` results in the following list of its contents:

```
~/fission $ ls ❶  
applications/ heat-production.txt neutron-release.txt ❷
```

- ❶ In the *fission* directory within her home directory, Lise types `ls` and then presses Enter.
- ❷ The shell responds by listing the contents of the current directory.

When she lists the contents, she sees that there are two files and one subdirectory. In the shell, directories may be rendered in a different color than files or may be indicated with a forward slash (/) at the end of their name, as in the preceding example.

Lise can also provide an argument to the `ls` command. To list the contents of the *applications* directory without entering it, she can execute:

```
~/fission $ ls applications ❶  
power/ propulsion/ weapons/ ❷
```

- ❶ Lise lists the contents of the *applications* directory without leaving the *fission* directory.
- ❷ The shell responds by listing the three directories contained in the *applications* directory.

The `ls` command can inform Lise about the contents of directories in her filesystem. However, to actually navigate to any of these directories, Lise will need the command `cd`.

Changing Directories (cd)

Lise can change directories with the `cd` command. When she types only those letters, the `cd` command assumes she wants to go to her home directory, so that's where it takes her:

```
~/fission $ cd ❶  
~ $
```

- ❶ Change directories to the default location, the home directory!

As you can see in this example, executing the `cd` command with no arguments results in a new prompt. The prompt reflects the new current working directory, home (~). To double-check, `pwd` can be executed and the home directory will be printed as an absolute path:

```
~ $ pwd ❶  
/filespace/people/l/lisemeitner ❷
```

- ❶ Print the working directory.

- ② The shell responds by providing the absolute path to the current working directory.

However, the `cd` command can also be customized with an **argument**, a parameter that follows the command to help dictate its behavior:

```
~/fission $ cd [path]
```

If Lise adds a space followed by the path of another directory, the shell navigates to that directory. The argument can be either an absolute path or a relative path.



Angle and Square Bracket Conventions

Using `<angle brackets>` is a common convention for terms that must be included and for which a real value must be substituted. You should not type in the less-than (`<`) and greater-than (`>`) symbols themselves. Thus, if you see `cd <argument>`, you should type in something like `cd mydir`. The `[square brackets]` convention denotes optional terms that may be present. Likewise, if they do exist, do not type in the `[` or `]`. Double square brackets (`[[]]`) are used to denote optional arguments that are themselves dependent on the existence of other `[optional]` arguments.

In the following example, Lise uses an absolute path to navigate to a sub-subdirectory. This changes the current working directory, which is visible in the prompt that appears on the next line:

```
~ $ cd /filesystem/people/l/lisemeitner/fission ①  
~/fission $ ②
```

- ① Lise uses the full, absolute path to the *fission* directory. This means, “change directories into the root directory, then the *filesystem* directory, then the *people* directory, and so on until you get to the *fission* directory.” She then presses Enter.
- ② She is now in the directory *~/fission*. The prompt has changed accordingly.

Of course, that is a lot to type. We learned earlier that the shorthand `~` means “the absolute path to the home directory.” So, it can be used to shorten the absolute path, which comes in handy here, where that very long path can be replaced with *~/fission*:

```
~/ $ cd ~/fission ①  
~/fission $
```

- ① The tilde represents the home directory, so the long absolute path can be shortened, accomplishing the same result.

Another succinct way to provide an argument to `cd` is with a **relative path**. A relative path describes the location of a directory *relative* to the location of the current directory. If the directory where Lise wants to move is inside her current directory, she can drop everything up to and including the current directory's name. Thus, from the *fission* directory, the path to the *applications* directory is simply its name:

```
~/fission $ cd applications ❶  
~/fission/applications $
```

- ❶ The *applications* directory must be present in the current directory for this command to succeed.

If a directory does not exist, bash will not be able to change into that location and will report an error message, as seen here. Notice that bash stays in the original directory, as you might expect:

```
~/fission $ cd biology  
-bash: cd: biology: No such file or directory  
~/fission $
```

Another useful convention to be aware of when forming relative paths is that the current directory can be represented by a single dot (`.`). So, executing `cd ./power` is identical to executing `cd power`:

```
~/fission/applications/ $ cd ./power ❶  
~/fission/applications/power/ $
```

- ❶ Change directories into this directory, then into the *power* directory.

Similarly, the parent of the current directory's parent is represented by two dots (`..`). So, if Lise decides to move back up one level, back into the *applications* directory, this is the syntax she could use:

```
~/fission/applications/power/ $ cd ..  
~/fission/applications/ $
```

Using the two-dots syntax allows relative paths to point anywhere, not just at subdirectories of your current directory. For example, the relative path `../../..` means three directories *above* the current directory.



Exercise: Change Directories

1. Open the Terminal.
2. Type `cd ..` at the command prompt and press Enter to move from your home directory to the directory above it.
3. Move back into your home directory using a relative path.
4. If you have downloaded Lise's directory tree from [the book's GitHub repository](#), can you navigate to that directory using what you know about `ls`, `cd`, and `pwd`?

A summary of a few of these path-generating shortcuts is listed in [Table 1-2](#).

Table 1-2. Path shortcuts

Syntax	Meaning
/	The root, or top-level, directory of the filesystem (also used for separating the names of directories in paths)
~	The home directory
.	This directory
..	The parent directory of this directory
../..	The parent directory of the parent directory of this directory

While seeing the names of files and directories is helpful, the content of the files is usually the reason to navigate to them. Thankfully, the shell provides myriad tools for this purpose. In the next section, we'll learn how to inspect that content once we've found a file of interest.

File Inspection (head and tail)

When dealing with input and output files for scientific computing programs, you often only need to see the beginning or end of the file (for instance, to check some important input parameter or see if your run completed successfully). The command `head` prints the first 10 lines of the given file:

```
~/fission/applications/power $ head reactor.txt  
  
# Fission Power Idea
```

The heat from the fission reaction could be used to heat fluids. In the same way that coal power starts with the production heat which turns water to steam and spins a turbine, so too nuclear fission might heat fluid that pushes a turbine. If somehow there were a way to have many fissions in one small space, the heat from those fissions could be used to heat quite a lot of water.

As you might expect, the `tail` command prints the last 10:

```
~/fission/applications/power $ head reactor.txt
```

```
the same way that coal power starts with the production heat which
turns water to steam and spins a turbine, so too nuclear fission
might heat fluid that pushes a turbine. If somehow there were a way to
have many fissions in one small space, the heat from those fissions
could be used to heat quite a lot of water.
```

Of course, it would take quite a lot of fissions.

Perhaps Professors Rutherford, Curie, or Fermi have some ideas on this topic.



Exercise: Inspect a File

1. Open a terminal program on your computer.
2. Navigate to a text file.
3. Use `head` and `tail` to print the first and last lines to the terminal.

This ability to print the first and last lines of a file to the terminal output comes in handy when inspecting files. Once you know how to do this, the next tasks are often creating, editing, and moving files.

Manipulating Files and Directories

In addition to simply finding files and directories, the shell can be used to act on them in simple ways (e.g., copying, moving, deleting) and in more complex ways (e.g., merging, comparing, editing). We'll explore these tasks in more detail in the following sections.

Creating Files (`nano`, `emacs`, `vi`, `cat`, `>`, and `touch`)

Creating files can be done in a few ways:

- With a graphical user interface (GUI) outside the terminal (like Notepad, Eclipse, or the IPython Notebook)

- With the `touch` command
- From the command line with `cat` and **redirection** (`>`)
- With a sophisticated text editor inside the terminal, like *nano*, *emacs*, or *vi*

Each has its own place in a programming workflow.

GUIs for file creation

Readers of this book will have encountered, at some point, a graphical user interface for file creation. For example, Microsoft Paint creates *.bmp* files and word processors create *.doc* files. Even though they were not created in the terminal, those files are (usually) visible in the filesystem and can be manipulated in the terminal. Possible uses in the terminal are limited, though, because those file types are not plain text. They have *binary* data in them that is not readable by a human and must be interpreted through a GUI.

Source code, on the other hand, is written in plain-text files. Those files, depending on the conventions of the language, have various filename extensions. For example:

- *.cc* indicates C++
- *.f90* indicates Fortran90
- *.py* indicates Python
- *.sh* indicates bash

Despite having various extensions, source code files are plain-text files and should not be created in a GUI (like Microsoft Word) unless it is intended for the creation of plain-text files. When creating and editing these source code files in their language of choice, software developers often use interactive development environments (IDEs), specialized GUIs that assist with the syntax of certain languages and produce plain-text code files. Depending on the code that you are developing, you may decide to use such an IDE. For example, MATLAB is the appropriate tool for creating *.m* files, and the IPython Notebook is appropriate for creating *.ipynb* files.

Some people achieve enormous efficiency gains from IDEs, while others prefer tools that can be used for any text file without leaving the terminal. The latter type of text editor is an essential tool for many computational scientists—their hammer for every nail.

Creating an empty file (`touch`)

A simple, empty text file, however, can be created with a mere “touch” in the terminal. The `touch` command, followed by a filename, will create an empty file with that name.

Suppose Lise wants to create a file to act as a placeholder for a new idea for a nuclear fission application, like providing heat sources for remote locations such as Siberia. She can create that file with the touch command:

```
~/fission/applications $ touch remote_heat.txt
```

If the file already exists, the touch command does no damage. All files have **metadata**, and touch simply updates the file's metadata with a new "most recently edited" time-stamp. If the file does not already exist, it is created.



Note how the *remote_heat.txt* file's name uses an underscore instead of a space. This is because spaces in filenames are error-prone on the command line. Since the command line uses spaces to separate arguments from one another, filenames with spaces can confuse the syntax. Try to avoid filenames with spaces. If you can't avoid them, note that the escape character (\) can be used to alert the shell about a space. A filename with spaces would then be referred to as *my\ file\ with\ spaces\ in\ its\ name.txt*.

While the creation of empty files can be useful sometimes, computational scientists who write code do so by adding text to code source files. For that, they need text editors.

The simplest text editor (cat and >)

The simplest possible way, on the command line, to add text to a file without leaving the terminal is to use a program called cat and the shell syntax >, which is called redirection.

The cat command is meant to help concatenate files together. Given a filename as its argument, cat will print the full contents of the file to the terminal window. To output all content in *reactor.txt*, Lise could use cat as follows:

```
~/fission/applications/power $ cat reactor.txt
```

```
# Fission Power Idea
```

```
The heat from the fission reaction could be used to heat fluids. In
the same way that coal power starts with the production heat which
turns water to steam and spins a turbine, so too nuclear fission
might heat fluid that pushes a turbine. If somehow there were a way to
have many fissions in one small space, the heat from those fissions
could be used to heat quite a lot of water.
```

```
Of course, it would take quite a lot of fissions.
```

```
Perhaps Professors Rutherford, Curie, or Fermi have some ideas on this topic.
```

This quality of `cat` can be combined with **redirection** to push the output of one file into another. Redirection, as its name suggests, *redirects* output. The greater-than symbol, `>`, is the syntax for redirection. The arrow collects any output from the command preceding it and redirects that output into whatever file or program follows it. If you specify the name of an existing file, its contents will be overwritten. If the file does not already exist, it will be created. For example, the following syntax pushes the contents of *reactor.txt* into a new file called *reactor_copy.txt*:

```
~/fission/applications/power $ cat reactor.txt > reactor_copy.txt
```

Without any files to operate on, `cat` accepts input from the command prompt.

Killing or Interrupting Programs

In the exercise above, you needed to use `Ctrl-d` to escape the `cat` program. This is not uncommon. Sometimes you'll run a program and then think better of it, or, even more likely, you'll run it incorrectly and need to stop its execution. `Ctrl-c` will usually accomplish this for noninteractive programs. Interactive programs (like `less`) typically define some other keystroke for killing or exiting the program. `Ctrl-d` will normally do the trick in these cases.

As an example of a never-terminating program, let's use the `yes` program. If you call `yes`, the terminal will print `y` ad infinitum. You can use `Ctrl-c` to make it stop.

```
~/fission/supercritical $ yes  
y  
y  
y  
y  
y  
y  
y  
y  
y  
Ctrl-c
```



Exercise: Learn About a Command

1. Open a terminal.
2. Type `cat` and press `Enter`. The cursor will move to a blank line.
3. Try typing some text. Note how every time you press `Enter`, a copy of your text is repeated.
4. To exit, type `Ctrl-d`. That is, hold down the `Control` key and press the lowercase *d* key at the same time.

Used this way, `cat` reads any text typed into the prompt and emits it back out. This quality, combined with redirection, allows you to push text into a file without leaving the command line. Therefore, to insert text from the prompt into the *remote_heat.txt* file, the following syntax can be used:

```
~fission/applications/power $ cat > remote_heat.txt
```

After you press Enter, the cursor will move to a blank line. At that point, any text typed in will be inserted into *remote_heat.txt*. To finish adding text and exit `cat`, type Ctrl-d.



Be careful. If the file you redirect into is not empty, its contents will be erased before it adds what you're writing.

Using `cat` this way is the simplest possible way to add text to a file. However, since `cat` doesn't allow the user to go backward in a file for editing, it isn't a very powerful text editor. It would be incredibly difficult, after all, to type each file perfectly the first time. Thankfully, a number of more powerful text editors exist that can be used for much more effective text editing.

More powerful text editors (nano, emacs, and vim)

A more efficient way to create and edit files is with a text editor. Text editors are programs that allow the user to create, open, edit, and close plain-text files. Many text editors exist. *nano* is a simple text editor that is recommended for first-time users. The most common text editors in programming circles are *emacs* and *vim*; these provide more powerful features at the cost of a sharper learning curve.

Typing the name of the text editor opens it. If the text editor's name is followed by the name of an existing file, that file is opened with the text editor. If the text editor's name is followed by the name of a nonexistent file, then the file is created and opened.

To use the *nano* text editor to open or create the *remote_heat.txt* file, Lise Meitner would use the command:

```
~fission/applications/power $ nano remote_heat.txt
```

Figure 1-3 shows the *nano* text editor interface that will open in the terminal. Note that the bottom of the interface indicates the key commands for saving, exiting, and performing other tasks.

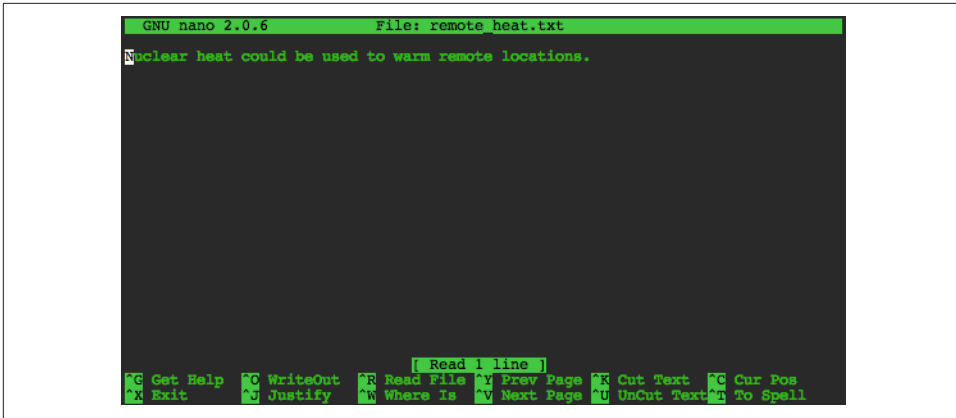


Figure 1-3. The nano text editor

If Lise wanted to use the *vim* text editor, she could use either the command `vim` or the command `vi` on the command line to open it in the same way. On most modern Unix or Linux computers, `vi` is a short name for `vim` (*vim* is *vi*, *improved*). To use *emacs*, she would use the `emacs` command.

Choose an Editor, Not a Side

A somewhat religious war has raged for decades in certain circles on the topic of which text editor is superior. The main armies on this battlefield are those that herald *emacs* and those that herald *vim*. In this realm, the authors encourage the reader to maintain an attitude of radical acceptance. In the same way that personal choices in lifestyle should be respected unconditionally, so too should be the choice of text editor. While the selection of a text editor can powerfully affect one's working efficiency and enjoyment while programming, the choice is neither permanent nor an indication of character.

Because they are so powerful, many text editors have a steep learning curve. The many commands and key bindings in a powerful text editor require practice to master. For this reason, readers new to text editors should consider starting with *nano*, a low-powered text editor with a shallower learning curve.



Exercise: Open nano

1. Open the Terminal.
2. Execute the command `nano`.
3. Add some text to the file.
4. Use the instructions at the bottom of the window to name and save the file, then exit *nano*.

Copying and Renaming Files (`cp` and `mv`)

Now that we've explored how to create files, let's start learning how to move and change them. To make a copy of a file, use the `cp` command. The `cp` command has the syntax `cp <source> <destination>`. The first required argument is the source file (the one you want to make a copy of), as a **relative** or **absolute path**. The second is the destination file (the new copy itself), as a relative or absolute path:

```
~/fission/applications/power $ ls
reactors.txt
~/fission/applications/power $ cp reactors.txt heaters.txt
~/fission/applications/power $ ls
reactors.txt heaters.txt
```

However, if the destination is in another directory, the named directory must already exist. Otherwise, the `cp` command will respond with an error:

```
~/fission/applications/power $ cp ./reactors.txt ./electricity/power-plant.txt
cp: cannot create regular file './electricity/power-plant.txt':
No such file or directory
```

If Lise doesn't need to keep the original file during a copy, she can use `mv` (move), which renames the file instead of copying it. The command evokes "move" because if the new name is a path in another directory, the file is effectively moved there.

Suppose that when browsing through her ideas, Lise notices an idea for a nuclear plane in the *propulsion* directory:

```
~/fission/applications/propulsion $ ls
nuclear_plane.txt
```

It really was not such a good idea, actually. A nuclear plane would probably be too heavy to ever fly. She decides to rename the idea, as a warning to others. It should be called *bad_idea.txt*. The `mv` command accepts two arguments: the original file path followed by the new file path. She renames *nuclear_plane.txt* to *bad_idea.txt*:

```
~/fission/applications/propulsion $ mv nuclear_plane.txt bad_idea.txt ❶
~/fission/applications/propulsion $ ls ❷
bad_idea.txt ❸
~/fission/applications/propulsion $ mv ./bad_idea.txt ../ ❹
```



```
~/fission/applications/propulsion $ ls .. ❺  
bad_idea.txt power/ propulsion/ weapons/ ❻
```

- ❶ Move (rename) *nuclear_plane.txt* to *bad_idea.txt*.
- ❷ Show the resulting contents of the directory.
- ❸ Indeed, the file is now called *bad_idea.txt*.
- ❹ Now, try moving *bad_idea.txt* to the *applications* directory.
- ❺ List the contents of the *applications* directory to see the result.
- ❻ The renamed file is now located in the *applications* directory above the *propulsion* directory.

Once all of her files have been properly named, Lise may need new directories to reorganize them. For this, she'll need the `mkdir` command.

Making Directories (`mkdir`)

You can make new directories with the `mkdir` (make directory) command. Using our usual path conventions, you can make them anywhere, not just in your current working directory. When considering a new class of theories about the nucleus, Lise might decide to create a directory called *nuclear* in the *theories* directory. The `mkdir` command creates a new directory at the specified path:

```
~/theories $ mkdir nuclear
```

The path can be relative or absolute. In order to create a new directory within the new *nuclear* directory, she can specify a longer path that delves a few levels deep:

```
~/theories $ mkdir ./nuclear/fission
```

Note, however, that the rule about not putting a file in a nonexistent directory applies to new directories too:

```
~/theories/nuclear $ mkdir ./nuclear/fission/uranium/neutron-induced  
mkdir: cannot create directory './nuclear/uranium/neutron-induced':  
No such file or directory
```

Making directories like this on the command line speeds up the process of organization and reduces the overhead involved. Of course, sometimes you may make a file or directory by mistake. To fix this, you'll need the `rm` command.

Deleting Files and Directories (`rm`)

Files and directories can be deleted using the `rm` (remove) command. Recall that there was a bad idea in the *applications* directory:

```
~/fission/applications $ ls
bad_idea.txt power/ propulsion/ weapons/
```

After some time, Lise might want to delete that bad idea file entirely. To do so, she can use the `rm` command. Given a path to a file, `rm` deletes it:

```
~/fission/applications $ rm bad_idea.txt
```

Once it's removed, she can check again for its presence with the `ls` command. As you can see, it has disappeared:

```
~/fission/applications $ ls
power/ propulsion/ weapons/
```

Note that once a file is removed, it is gone forever. There is no safety net, no trash can, and no recycling bin. Once you delete something with `rm`, it is truly gone.



Be very careful when using `rm`. It is permanent. With `rm`, recall the adage “Measure twice, cut once.” Before using `rm`, consciously consider whether you really want to remove the file.

Since propulsion with nuclear heat, in general, seems unlikely given the weight, Lise may decide to delete the *propulsion* directory entirely. However, if she just provides the path to the directory, the `rm` command returns an error, as shown here:

```
~/fission/applications $ rm propulsion
rm: propulsion: is a directory
```

This error is a safety feature of `rm`. To delete directories, it is necessary to use the `-r` (recursive) flag. Flags such as `-r` modify the behavior of a command and are common in the shell. This flag tells `rm` to descend into the directory and execute the command all the way down the tree, deleting all files and folders below *propulsion*:

```
~/fission/applications $ rm -r propulsion
```

This requirement prevents you from deleting entire branches of a directory tree without confirming that you do, in fact, want the shell to descend into all subdirectories of the given directory and delete them, as well as their contents.

On some platforms, just to be safe, the `rm` command requests confirmation at each new subdirectory it encounters. Before it deletes a subdirectory, it will ask: “rm: descend into directory ‘subdirectoryname’?” Type `y` or `n` to confirm “yes” or “no,” respectively. This can be avoided if an `f` (for force) is added to the flags. The command to force removal of a directory and all its subdirectories is `rm -rf <directory name>`.



While `rm -rf` can be used carefully to great effect, never execute `rm -rf *`. Unscrupulous mischief-makers may recommend this, but it will have catastrophic consequences. Do not fall for this tomfoolery.

The next section will cover some examples of more flags available to commands in the shell.



Exercise: Make and Remove Files and Directories

1. Open the Terminal.
2. Use `mkdir` to create a directory with a few empty subdirectories.
3. Use `touch` to create five empty files in those directories, and use `ls` to inspect your work.
4. With one command (hint: it will have to be recursive), remove the whole directory. Do you need to use the force flag to avoid typing `y` repeatedly?

Flags and Wildcards

Flags are often important when using these file and directory manipulation commands. For instance, you can `mv` a directory without any flags. However, copying a directory without the recursive flag fails. Let's look at an example. Since all applications generating power start by generating heat, a new directory called *heat* could start as a duplicate of the *power* directory:

```
~/fission/applications $ cp power/ heat/  
cp: omitting directory `power/'
```

The copy command, not accepting a directory as a valid copy target, throws the error “cp: omitting directory *directoryname*”. To copy the directory and its contents with `cp`, the `-r` (recursive) flag is necessary:

```
~/fission/applications $ cp -r power/ heat/
```

An alternative to copying, moving, or removing entire directories is to use a wildcard character to match more than one file at once. In the bash shell, the asterisk (*) is a wildcard character. We'll talk about this more in [Chapter 8](#); for now, just note that the asterisk means, approximately, *match everything*.

In the following example, all the files in the directory are matched by the asterisk. Those files are *all* copied into the destination path:

```

~ $ cp beatles/* brits/
~ $ cp zeppelin/* brits/
~ $ cp beatles/john* johns/
~ $ cp zeppelin/john* johns/
~ $ ls brits
george jimmy john john_paul paul ringo robert
~ $ ls johns
john john_paul

```

But notice that we’ve overwritten a “john” during the second copy into each directory. To help avoid making such mistakes, you can use `-i` to run the command interactively; the shell will then ask you to confirm any operations it thinks seem suspicious:

```

~ $ cp beatles/john* johns/.
~ $ cp -i beatles/john* johns/.
cp: overwrite `johns/./john'? y

```

In a sense, `-i` is the opposite of `-f`, which forces any operations that the shell might otherwise warn you about:

```

~ $ mv zeppelin/john deceased/.
~ $ mv beatles/john deceased/.
mv: overwrite `deceased/./john'? n
~ $ mv -f beatles/john deceased/.

```

In this section, we have covered a few flags commonly available to commands on the command line. However, we have only scratched the surface. Most available commands possess many customized behaviors. Since there are far too many to memorize, the following section discusses how to get help and find more information about commands.

Getting Help

Now that you have become familiar with the basics, you can freely explore the terminal. The most important thing to know before venturing forth, however, is how to get help.

Reading the Manual (man)

The program `man` (manual) is an interface to online reference manuals. If you pass the name of a command or program to `man` as an argument, it will open the help file for that command or program. To determine what flags and options are available to the `ls` command, then, typing `man ls` would provide the instructions for its use. Since `man` is itself a program, we can type `man man` to get the instructions for using `man`:

```

~ $ man man

NAME
    man - an interface to the on-line reference manuals

```

SYNOPSIS

```
man [-c|-w|-tZ] [-H[browser]] [-T[device]] [-adhu7V]
[-i|-I] [-m system[,...]] [-L locale] [-p string] [-C
file] [-M path] [-P pager] [-r prompt] [-S list] [-e
extension] [[section] page ...] ...
man -l [-7] [-tZ] [-H[browser]] [-T[device]] [-p
string] [-P pager] [-r prompt] file ...
man -k [apropos options] regexp ...
man -f [whatis options] page ...
```

DESCRIPTION

`man` is the systems manual pager. Each page argument given to `man` is normally the name of a program, utility or function. The manual page associated with each of these arguments is then found and displayed. A section, if provided, will direct `man` to look only in that section of the manual. The default action is to search in all of the available sections, following a pre-defined order and to show only the first page found, even if page exists in several sections.

<snip>

What follows `man` in the SYNOPSIS is a listing of the optional and required arguments, options, and variables.

Arguments, options, and variables

In these `man` pages, you'll see that there are different ways to pass information to the command-line programs and commands you need to use. We've seen the first one: *arguments*. An argument simply gets added after the command. You can add multiple arguments if the command expects that behavior. We've added single arguments when we've changed into a specific directory (e.g., `cd .`). We also used two arguments at once with `cp` (e.g., `cp <source> <destination>`). We also saw, for example, that the `ls` command with the single argument `.` lists the contents of the current directory:

```
~/weaponry $ ls .
fear  ruthless_efficiency  surprise
```

We've also seen *options*, also called *flags* or *switches* (e.g., the recursive flag, `-r`). These tell the program to run in some predefined way. Options are usually specified with a minus sign (`-`) in front of them. For instance, if we run `man ls` and scroll down, we see that the `-r` option lists directory contents in reverse order. That is:

```
~/weaponry $ ls -r .
surprise  ruthless_efficiency  fear
```



Be careful—flags (like `-r`) don't necessarily have the same meaning for every command. For many commands, `-r` indicates *recursive* behavior, but for `ls`, it prints the directory contents in *reverse* order.

Variables can be used to pass in specific kinds of information and are usually specified with a double minus sign (`--`, typically pronounced “minus minus” or “dash dash”). Further perusal of the `ls` `man` page indicates that a variable called `sort` can be set to certain values to sort directory contents in various ways. To provide a value to `sort`, we use an equals sign (`=`). For instance, `--sort=time` sorts directory contents by file modification time, with the most recent file first:

```
~/weaponry $ ls --sort=time .  
fear  surprise  ruthless_efficiency
```

All of the arguments, options, and variables for a command are detailed in the `man` page for that command. To see how they are used, you will need to scroll down in the `man` page document to where they are explained. To scroll down, it's helpful to know how to use `less`.

Moving around in `less`

`man` opens the help documents in a program called `less`, which you can use to look at other text files as well (just call `less [filename]`). There's lots to learn about `less` (use `man less` to get an overview), but the most important things to know are as follows:

- Use the up and down arrows to scroll up and down.
- Use Page Up and Page Down (or the space bar) to move up or down by an entire page.
- Use a forward slash (`/`) followed by a search term and then Enter to search for a particular word. The letter `n` (next) toggles through each occurrence.
- Use `h` to display help inside `less`—this displays all the possible commands that `less` understands.
- Use `q` to quit.

`less` is modeled on an earlier program called `more`. However, `more` has fewer features, and you probably shouldn't bother with it. So, always remember: *less is more*.



Exercise: Use the man Pages with less

1. Open the Terminal.
2. Use the `man` command and the preceding notes on `less` to learn about the commands covered already in this chapter (e.g., `mkdir`, `touch`, `mv`, `cp`, etc.)

Of course, before you can use `man` and `less` to find information about available commands, you must know what commands are available. For that, we need a command called `apropos`.

Finding the Right Hammer (`apropos`)

The bash shell has so many built-in programs, practically no one has all of their names memorized. Since the `man` page is only helpful if you know the name of the command you're looking for, you need some tool to determine what that command is. Thankfully, this tool exists. You can search the `man` pages for keywords with a command called `apropos`. Let's say you want to know what text editors are available. You might search for the string "text editor":

```
~ $ apropos "text editor" ❶
ed(1), red(1)             - text editor ❷
vim(1)                    - Vi IMproved, a programmers text editor ❸
```

- ❶ To search for an installed command based on a keyword string, use `apropos`.
- ❷ `ed` and `red` show up together, because their full description is "text editor."
- ❸ `vim` appears next, with its longer description. Other installed editors will not appear if the exact phrase "text editor" does not appear in their `man` pages. What happens if you try `apropos editor`?

An optimistic physicist, Lise might have been curious enough to query physics-related commands. Unfortunately, she might be disappointed to find there aren't many:

```
~ $ apropos physics
physics: nothing appropriate
```



Exercise: Find and Learn About a Command

1. Open the Terminal.
2. Search your computer for commands by using `apropos` and a keyword.
3. Take some time to explore the `man` page of a command we've discussed or of another command or program you know of. Learn about a couple of new arguments or options and try them out. Practice killing or interrupting programs if necessary.

Now that this chapter has touched on the various commands for running processes and manipulating files, let's see how those commands can be combined into powerful pipelines using redirection and pipes.

Combining Utilities with Redirection and Pipes (`>`, `>>`, and `|`)

The power of the shell lies in the ability to combine these simple utilities into more complex algorithms very quickly. A key element of this is the ability to send the output from one command into a file or to pass it directly to another program.

To send the output of a command into a file, rather than printing it to the screen as usual, redirection is needed. A text or data stream generated by the command on the lefthand side of the arrow is sent (redirected) into the file named on the righthand side. One arrow (`>`) will create a new file or overwrite the contents of an existing one with the stream provided by the lefthand side. However, two arrows (`>>`) will append the stream to the end of an existing file, rather than overwriting it. If Lise wants to create a new file containing only the first line of another, she can combine the `head` command and the redirection method to achieve this in one line:

```
~/fission/applications/power $ head -1 reactor.txt > reactor_title.txt
```

Now, the content of *reactor_title.txt* is simply:

```
# Fission Power Idea
```

To chain programs together, the pipe (`|`) command can be used in a similar fashion. The output of one program can be used as the input of another. For example, to print the middle lines of a file to the screen, `head` and `tail` can be combined. To print only line 11 from the *reactor.txt* file, Lise can use `head`, `tail`, and a pipe:

```
~/fission/applications/power $ head -1 reactor.txt | tail -1  
Of course, it would take quite a lot of fissions.
```

With these methods, any program that reads lines of text as input and produces lines of text as output can be combined with any other program that does the same.

Now that you've seen how the many simple commands available in the shell can be combined into ad hoc pipelines, the incredible combinatoric algorithmic power of the shell is at your fingertips—but only if you have the right permissions.

Permissions and Sharing

Permissions are a subtle but important part of using and sharing files and using commands on Unix and Linux systems. This topic tends to confuse people, but the basic gist is that different people can be given different types of access to a given file, program, or computer.

At the highest level, the filesystem is only available to users with a user account on that computer. Based on these permissions, some commands allow users to connect to other computers or send files. For example:

- `ssh [user@host]` connects to another computer.
- `scp [file] [user@host]:path` copies files from one computer to another.

Those commands only work if the user issuing them has *permission* to log into the filesystem. Otherwise, he will not be able to access the file system at all.

Once they have accessed a computer's filesystem, however, different types of users may have different types of access to the various files on that system. The “different types of people” are the individual *user* (u) who owns the file, the *group* (g) who's been granted special access to it, and *all others* (o). The “different types of access” are permission to *read* (r), *write* (w), or *execute* (x) a file or directory.

This section will introduce three commands that allow you to manage the permissions of your files:

- `ls -l [file]` displays, among other things, the permissions for that file.
- `chown [-R] [[user]][:group] target1 [[target2 ...]]` changes the individual user and group ownership of the target(s), recursively if -R is used and one or more targets are directories.
- `chmod [options] mode[,mode] target1 [[target2 ...]]` changes or sets the permissions for the given target(s) to the given mode(s).

The first of these, `ls -l`, is the most fundamental. It helps us find out what permission settings apply to files and directories.

Seeing Permissions (ls -l)

We learned earlier in this chapter that `ls` lists the contents of a directory. When you explored the man page for `ls`, perhaps you saw information about the `-l` flag, which

lists the directory contents in the “long format.” This format includes information about permissions.

Namely, if we run `ls -l` in a directory in the filesystem, the first thing we see is a code with 10 permission digits, or “bits.” In her *fission* directory, Lise might see the following “long form” listing. The first 10 bits describe the permissions for the directory contents (both files and directories):

```
~/fission $ ls -l
drwxrwxr-x  5 lisemeitner  expmt  170 May 30 15:08 applications
-rw-rw-r--  1 lisemeitner  expmt   80 May 30 15:08 heat-generation.txt
-rw-rw-r--  1 lisemeitner  expmt   80 May 30 15:08 neutron-production.txt
```

The first bit displays as a `d` if the target we’re looking at is a directory, an `l` if it’s a link, and generally `-` otherwise. Note that the first bit for the *applications* directory is a `d`, for this reason.

To see the permissions on just one file, the `ls -l` command can be followed by the filename:

```
~/fission $ ls -l heat-generation.txt
-rw-rw-r--  1 lisemeitner  expmt   80 May 30 15:08 heat-generation.txt
```

In this example, only the permissions of the desired file are shown. In the output, we see one dash followed by three sets of three bits for the *heat-generation.txt* file (`-rw-rw-r--`). Let’s take a look at what this means:

- The first bit is a dash, `-`, because it is not a directory.
- The next three bits indicate that the user owner (`lisemeitner`) can read (`r`) or write (`w`) this file, but not execute it (`-`).
- The following three bits indicate the same permissions (`rw-`) for the group owner (`expmt`).
- The final three bits (`r--`) indicate read (`r`) but not write or execute permissions for everyone else.

All together, then, Lise (`lisemeitner`) and her Experiment research group (`expmt`) can read or change the file. They cannot run the file as an executable. Finally, other users on the network can only read it (they can never write to or run the file).

Said another way, the three sets of three bits indicate permissions for the user owner, group owner, and others (in that order), indicating whether they have read (`r`), write (`w`), or execute (`x`) privileges for that file.

The `ls` man page provides additional details on the rest of the information in this display, but for our purposes the other relevant entries here are the two names that fol-

low the permission bits. The first indicates that the user `lisemeitner` is the individual owner of this file. The second says that the group `expmt` is the group owner of the file.



Exercise: View the Permissions of Your Files

1. Open a terminal.
2. Execute `ls -l` on the command line. What can you learn about your files?
3. Change directories to the `/` directory (try `cd /`). What are the permissions in this directory? What happens if you try to create an empty file (with `touch <filename>`) in this directory?

In addition to just observing permissions, making changes to permissions on a file system is also important.

Setting Ownership (`chown`)

It is often helpful to open file permissions up to one's colleagues on a filesystem. Suppose Lise, at the Kaiser Wilhelm Institute, wants to give all members of the institute permission to read and write to one of her files, *heat-generation.txt*. If those users are all part of a group called `kwi`, then she can give them those permissions by changing the group ownership of the file. She can handle this task with `chown`:

```
~/fission $ chown :kwi heat-generation.txt
~/fission $ ls -l heat-generation.txt
-rw-rw-r-- 1 lisemeitner kwi 80 May 30 15:08 heat-generation.txt
```



Exercise: Change Ownership of a File

1. Open a terminal.
2. Execute the `groups` command to determine the groups that you are a part of.
3. Use `chown` to change the ownership of a file to one of the groups you are a part of.
4. Repeat step 3, but change the group ownership back to what it was before.

However, just changing the permissions of the file is not quite sufficient, because directories that are not executable by a given user can't be navigated into, and directories that aren't readable by a given user can't be printed with `ls`. So, she must also

make sure that members of this group can navigate to the file. The next section will show how this can be done.

Setting Permissions (chmod)

Lise must make sure her colleagues can visit and read the dictionary containing the file. Such permissions can be changed by using `chmod`, which changes the file *mode*. Since this is a directory, it must be done in recursive mode. If she knows her home directory can be visited by members of the `kwi` group, then she can set the permissions on the entire directory tree under `~/fission` with two commands. The first is again `chown`. It sets the *fission* directory's group owner (recursively) to be `kwi`:

```
~ $ chown -R :kwi fission/
```

Next, Lise changes the file mode with `chmod`. The `chmod` syntax is `chmod [options] <mode> <path>`. She specifies the recursive option, `-R`, then the mode to change the group permissions, adding (+) reading and execution permissions with `g+rx`:

```
~ $ chmod -R g+rx fission/
```

Many other modes are available to the `chmod` command. The mode entry `g+rx` means we *add* the read and execution bits to the group's permissions for the file. Can you guess the syntax for subtracting the group's read permissions? The manual page for `chmod` goes into exceptional detail about the ways to specify file permissions. Go there for special applications.

Physicists using large scientific computing systems rely heavily on permissions to securely and robustly share data files and programs with multiple users. All of these permissions tools are helpful with organizing files. Another tool available for organizing files across filesystems is the *symbolic link*.

Creating Links (ln)

The `ln` command allows a user to create a hard or symbolic link to a file or program. This effectively creates more than one reference pointing to where the contents of the file are stored. This section will focus on symbolic links rather than hard links.

Symbolic links are useful for providing access to large, shared resources on a networked filesystem. Rather than storing multiple copies of large datasets in multiple locations, an effective physicist can create symbolic links instead. This saves hard drive space, since a symbolic link only takes up a few bytes. Also, it saves time. Since the links can be placed in easy-to-find locations, colleagues will spend less time searching deeply nested subdirectories for desired programs and data files.

For our purposes, symbolic links (created with `ln -s`) are the safest and most useful. Let's say, for instance, that Lise has compiled a program that suggests a random pair of isotopes likely to result from a uranium fission. Her colleagues have a hard time

remembering whether the program is called *fission_fragments* or just *fragments*. When they try *fission_fragments*, bash responds with a warning—the command is not a valid path:

```
~/programs/fission $ ./fission_fragments
./fission_fragments: Command not found.
```

One solution is to add a symbolic link. A new link at the incorrect filename, pointing to the correct filename, can be created with the syntax `ln -s <source_path> <link_path>`:

```
~/programs/fission $ ln -s fragments fission_fragments
```

With that complete, a new symbolic link has been created. It can be viewed with `ls -l`, and appears to be just like any other file except for the arrow showing it is just a pointer to the *fragments* program:

```
~/programs/fission $ ls -l ❶
-rwxrwxr-x 1 lisemeitner staff 20 Nov 13 19:02 fragments ❷
lrwxrwxr-x 1 lisemeitner staff  5 Nov 13 19:03 fission_fragments -> fragments
```

- ❶ Input: Execute the “list in long form” command on the command line.
- ❷ Output: the file listing now shows both the *fragments* file and, on the next line, the *fission_fragments* file, with an arrow indicating that it is a symbolic link to the fragments executable. Note also that the first of the 10 permission bits for that file is an `l` for “link.”

Now, with this symbolic link in the directory, Lise’s colleagues can use either name with the same success. Furthermore, recall that a dot (.) stands for the current directory and that slashes (/) separate directory and file names. Therefore, *./myfile* refers to *myfile* in the current directory. When running a program in the current directory, you must include the dot-slash. As you can see from the following, this works equally well on symbolic links as it does on normal files:

```
~/programs/fission$ ./fission_fragments
140Xe 94Sr
```

Symbolic links are useful for providing access to large, shared resources, rather than storing multiple copies in multiple hard-to-reach locations. Another common way physicists gain access to large, shared resources is by accessing them on remote machines. We’ll discuss the power of connecting to other computers in the next section.

Connecting to Other Computers (ssh and scp)

This powerful feature of the command line, providing access to networked and remote filesystems, is key to high-performance computing tasks. Since most large

high-performance or high-throughput computing resources can only be accessed by SSH (Secure SHell) or similar protocols through the command line, truly high-powered computer systems simply are not accessible without use of the shell.

If you have the right credentials, you can even get access to another machine through the shell. You can do this using a program called *ssh*. For instance, for the user *grace* to log on to a networked computer *mk1*, she would use the *ssh* command with an argument specifying her username and the computer name, connected by the *@* symbol:

```
~ $ ssh grace@mk1
```

Or, if *mk1* is a computer located on the remote network domain *harvard.edu*, Grace can connect to that computer from her home computer with the full location of the computer in its domain:

```
~ $ ssh grace@mk1.harvard.edu
```

Once logged into the computer, Grace has access to the files and directories in the remote filesystem and can interact with them just as she does locally.

She can use the *scp* (secure copy) command to copy files and directories from one computer to another. It has the syntax *scp <source_file> [[user@]host]:<destination>*. So, to copy a *notes.txt* file from her local computer to the *COBOL* directory on the *mk1.harvard.edu* filesystem, she would execute:

```
~ $ scp ./notes.txt grace@mk1.harvard.edu:~/COBOL/notes.txt
```



Both *ssh* and *scp* require a valid username and password on the remote machine.

When she connects to another computer, Grace has access to its filesystem. On that system, there are not only different files, but also a different *environment*. We'll look at the environment and how it is configured next.

The Environment

In addition to providing commands, a filesystem hierarchy, and a syntax for navigation, the bash shell defines a computing environment. This computing environment can be customized using environment variables. We can investigate our environment with a program called *echo*. The *echo* command prints arguments to the terminal. In the case of a string argument, the string is printed verbatim:

```
~ $ echo "Hello World"
Hello World
```

In the case of environment variables, however, `echo` performs *expansion*, printing the values of the variables rather than just their names. You invoke these variables on the command line by prepending a `$` to the variable name.

When, in 1959, she began to design the first machine-independent programming language (COBOL), Grace Hopper did not have `bash`. `Bash`, after all, could never have come into existence without her breakthrough. Hypothetically, though, if she had had `bash`, her environment might have behaved liked this:

```
~ $ echo $USERNAME ❶
grace
~ $ echo $PWD
/filespace/people/g/grace ❷
```

- ❶ Echo the value of the `USERNAME` environment variable. On certain platforms, this variable is called `USER`.
- ❷ The computer stores the working directory in the environment variable `PWD`; the command `pwd` is simply a shortcut for `echo $PWD`.

Shell variables are replaced with their values when executed. In `bash`, you can create your own variables and change existing variables with the `export` command:

```
~ $ export GraceHopper="Amazing Grace"
```

Variables are case-sensitive. For this reason, the following command will successfully echo the assigned string:

```
~ $ echo $GraceHopper
Amazing Grace
```

However, none of the following will succeed:

```
~ $ echo GraceHopper
~ $ echo GRACEHOPPER
~ $ echo $GRACEHOPPER
```

Table 1-3 lists some of the most common and important shell variables. These variables often become essential for defining the computer’s behavior when the user compiles programs and builds libraries from the command line.

Table 1-3. Common environment variables

Variable name	Meaning
USER	User name
PATH	List of absolute paths that are searched for executables
PWD	Current directory (short for print working directory)

Variable name	Meaning
EDITOR	Default text editor
GROUP	Groups the user belongs to
HOME	Home directory
~	Same as HOME
DISPLAY	Used in forwarding graphics over a network connection
LD_LIBRARY_PATH	Like PATH, but for precompiled libraries
FC	Fortran compiler
CC	C compiler

Environment variables can be used to store information about the environment and to provide a shorthand for long but useful strings such as absolute paths. To see all of the environment variables that are active in your terminal session, use the `env` command. Rear Admiral Hopper might see something like:

```
~/fission $ env
SHELL=/bin/bash
USER=grace
EDITOR=vi
LD_LIBRARY_PATH=/opt/local/lib:/usr/local
PATH=/opt/local/lib:/filesystem/people/g/grace/anaconda/bin:/opt/local/bin
PWD=/filesystem/people/g/grace/languages
LANG=en_US.utf8
PWD=/filesystem/people/g/grace
LOGNAME=grace
OLDPWD=/filesystem/people/g/grace/languages/COBOL
```

To make an environment variable definition active every time you open a new terminal, you must add it to a file in your home directory. This file must be called `.bashrc`.

Saving Environment Variables (.bashrc)

A number of files in a bash shell store the environment variables that are active in each terminal session. They are plain-text files containing bash commands. These commands are executed every time a terminal window is opened. Thus, any environment variables set with `export` commands in those files are active for every new terminal session.

To configure and customize your environment, environment variables can be added or edited in `~/.bashrc`, the main user-level bash configuration file. The `export` com-

mands we executed in the terminal before added new environment variables for a single terminal session. To add or change an environment variable for every session, we use *.bashrc*.



The leading *.* in *.bashrc* makes the file a hidden file.

User-specific configuration exists in many files. In addition to the *.bashrc* file, you may see others, such as *.bash_profile* or, on newer Mac OS machines, *.profile*. Do any of those exist on your computer? If so, open that file and confirm that it contains the text `source ~/.bashrc`.



Exercise: Configure Your Shell with *.bashrc*

1. Use your text editor to open the *.bashrc* file in your home directory. If no such file exists, create it.
2. Add an export command to set a variable called *DATA* equal to the location of some data on your filesystem.
3. Open a new terminal window and query the *DATA* variable with `echo`.
4. What is the result of `cd $DATA`? Can you imagine ways this behavior could make navigating your files easier?

A new terminal instance will automatically reflect changes to the *.bashrc* file. However, the `source` command can be used to make changes to *.bashrc* take effect immediately, in the current session:

```
~ $ source .bashrc
```

The bash shell can be customized enormously with commands executed in the *.bashrc* file. This customization is ordinarily used to specify important paths and default behaviors, making the shell much more efficient and powerful. The most important variable in your *.bashrc* file is the *PATH*.

Running Programs (PATH)

Based on the environment, the shell knows where to find the commands and programs you use at the command line. Unless you modify your environment, you can't run just any old program on your computer from any directory. If you want to run a

program in a nonstandard location, you have to tell the shell exactly where that program is by invoking it with an **absolute** or **relative** Unix path.

For instance, in **Chapter 14**, we will learn to build a program. However, after we've done so, we can still only run that program if we tell the shell exactly where it is. With the programs we have seen so far, the name of the command is sufficient. However, because bash only searches certain locations for available commands, the `fragments` command will not be found:

```
~/programs/fission $ fragments ❶  
fragments: Command not found. ❷
```

- ❶ We attempt to run the *fragments* program.
- ❷ The shell's response indicates that it cannot find the named program (because it is not in the PATH).

Indeed, even in the proper directory, you must indicate the full path to the program by adding the leading dot-slash before the computer understands what program to run:

```
~/programs/fission $ ./fragments  
136Cs 99Tc
```

In order for the computer to find the *fragments* program without us typing the full path, the PATH environment variable must contain the directory holding the program. Without the full path, the bash shell only executes commands found when searching the directories specified in the PATH environment variable. To add this folder to the PATH environment variable, Lise can execute the following command:

```
~/programs $ export PATH=$PATH:/fileSPACE/people/l/lisemeitner/programs/fission
```

The first part of this command uses `export` to set the PATH variable. Everything on the righthand side of the equals sign will become the new PATH. The first element of that path is the old PATH variable value. The second element, after the colon, is the new directory to add to the list of those already in the PATH. It will be searched last.



Exercise: Customize Your PATH

1. In the terminal, use `echo` to determine the current value of the PATH environment variable. Why do you think these directories are in the PATH?
2. Use `export` to add your current directory to the end of the list. Don't forget to include the previous value of PATH.
3. Use `echo` once again to determine the new value.

Can you think of a way that the `PWD` environment variable could be used to shorten the preceding command? In addition to shortening commands and paths by setting environment variables, configuration files are an excellent place to permanently give shorter nicknames to other commands. In the next section, we'll see how to do this with the `alias` command.

Nicknaming Commands (`alias`)

In the same way that you can create variables for shortening long strings (like `$DATA`, the path to your data), you can create shorthand aliases for commands. `alias` does a simple replacement of the first argument by the second. If, for example, you like colors to distinguish the files and directories in your shell session, you'll always want to use the `--color` variable when calling `ls`. However, `ls --color` is quite a lot to type. It is preferable to reset the meaning of `ls` so that it behaves like `ls --color`. The `alias` command allows you to do just that. To replace `ls` with `ls --color`, you would type:

```
alias ls 'ls --color'
```

Once that command has been executed in the terminal, typing `ls` is equivalent to typing `ls --color`. Just like an environment variable, to make an alias active every time you open a new terminal, you must add this definition to your `.bashrc` file, which is executed upon login.



To keep their `.bashrc` files cleaner and more readable, many individuals choose to keep all of their aliases in a separate hidden file (called something like `.bash_aliases`). To load it, they include the following line in their `.bashrc` files:

```
source ~/.bash_aliases
```

Now that the `.bashrc` bash script has demonstrated the power of automating bash commands, the following section will show how to write your own bash scripts.

Scripting with Bash

Repeating processes on the command line is made easy with files (like `.bashrc`) that can store many commands and be executed at will. Any such series of commands can be placed into a file called a *script*. This type of file, like a program, can be written once and executed many times.

Bash scripts typically end in the `.sh` extension. So, the first step for creating a bash script is to create such a file. As we did earlier in this chapter, you can do this by opening a text editor like *nano* and supplying the filename as an argument:

```
~ $ nano explore.sh
```

Any commands that are valid in the terminal are valid in a bash script. Some text in a bash script might be for reference, however. This text, called a *comment*, must be denoted with a #.

If Lise would like to automate the process of exploring her directory tree, she might write a bash script for the task. A very simple bash script that enters three levels of parent directories and prints the contents as well as the directory names is only a few simple lines:

```
# explore.sh ❶
# explore the three directories above this one
# print a status message
echo "Initial Directory:"
# print the working directory
pwd
# list the contents of this directory
ls
echo "Parent Directory:"
# ascend to the parent directory
cd ..
pwd
ls
echo "Grandparent Directory:"
cd ..
pwd
ls
echo "Great-Grandparent Directory:"
cd ..
pwd
ls
```

- ❶ Comments are preceded by a # symbol. They are reference text only and are not executed.

After you save this file, there is only one more step required to make it a bona fide program. To run this script on the command line, its permissions must be set to *executable*. To make this the case, Lise must execute (in the terminal) the command:

```
~ $ chmod a+x explore.sh
```

Now, the *explore.sh* script is runnable. To run the command, Lise must either call it with its full path or add the location to her PATH environment variable. When we use a relative path, the execution looks like:

```
~ $ ./explore.sh
```



Exercise: Write a Simple Bash Script

1. Create a file called *explore.sh*.
2. Copy the example script into that file.
3. Change the permissions of the file so that it is executable.
4. Run it and watch the contents of your filesystem be printed to the terminal.

Much more sophistication is possible in a bash script, but that is somewhat beyond the scope of this chapter. To learn more about sophisticated bash scripting, check out some of the O'Reilly books on the topic or sign up for a workshop like those run by [Software Carpentry](#).



The history Command

At the end of some series of bash commands, an effective physicist may want to create a bash script to automate that set of commands in the future. The `history` command provides a list of all the most recent commands executed in the terminal session. It is very helpful for recalling recent work and enshrining it in a bash script.

Command Line Wrap-up

This chapter has only just scratched the surface of the power the command line holds. It has covered:

- Navigating the filesystem
- Creating, deleting, and moving files and directories
- Finding help
- Running commands
- Handling permissions
- Writing scripts

If you want to find out more about the command line, many books and Internet resources are available. Cameron Newham's *Learning the bash Shell* (O'Reilly) is a good place to start, and Software Carpentry's workshops and online resources provide an excellent introduction to the deeper secrets of bash.

Analysis and Visualization

Churning out terabytes of data from simulations or experiments does not, on its own, constitute science. Only analysis and visualization can transform raw data into true scientific insight. Unanalyzed data is merely data—only interpretation and communication can sufficiently illuminate and clarify the scientific meaning of results. When analysis and visualization succeed, compelling data becomes a convincing result.

There was an era in the physical sciences when data was collected in laboratory notebooks, and when the time to publish plots of that data came about, it was done by hand. Legend has it that this was sometimes begun on enormous sheets of graph paper on the wall of a lab and scaled down to a reasonable publishable size by big, slow scanning machines. Many physicists and mathematicians, Roger Penrose not least among them, continue to make plots and diagrams with pen and paper. Nonetheless, it is an increasingly lost art.

While it is tempting to feel a nostalgia for freehand drawings of the complex plane, this chapter should inspire you to embrace the future instead. This chapter will provide an overview of principles and tools for data preparation, analysis, and visualization appropriate for publication-quality results in the physical sciences. Finally, a few examples of analysis and visualization using Python tools will be addressed. This chapter will provide a taste of the analysis tools that will then be discussed in detail in Chapters 9, 10, and 11.

Preparing Data

Researchers encounter data in many formats, from many sources. They accordingly can spend significant effort loading, cleaning, collapsing, expanding, categorizing, and generally “munging” data into consistent formats that can be used for analysis and plotting. Some basic methods for retrieving data from files will be covered in

Chapter 10, as will more advanced data manipulation. Depending on the source of the data, advanced data munging for myriad formats may be necessary. A few will be discussed here.

Faced with imperfect data in one or more raw formats, the researcher must perform several steps before even beginning to analyze or visualize it:

- Load the data into an analysis-ready format.
 - Possibly convert the data to an intermediate data storage format (CSV, HDF5, SQL, FITS, ENDF, ENSDF).
 - Convert the data to an easy-access data structure (NumPy arrays, Pandas data frames).
- Clean the data.
 - Handle missing values.
 - Drop them from analysis.
 - Replace them with defaults.
 - Count them for statistics.
 - Fix mangled or erroneous entries.
 - Detect errors.
 - Sort disordered data.
 - Standardize data formats.
 - Handle stylistic issues.
 - Rename long or irregular fields.
 - Reformat irregularly formatted dates, times, numbers, strings, etc.
- Combine the data with metadata.
 - Populate results tables with additional/external data.
 - Add identifying numbers and dates, for provenance.
 - Merge results from independent detectors, etc.

A visual representation of the aspects of this process appears in **Figure 7-1**.

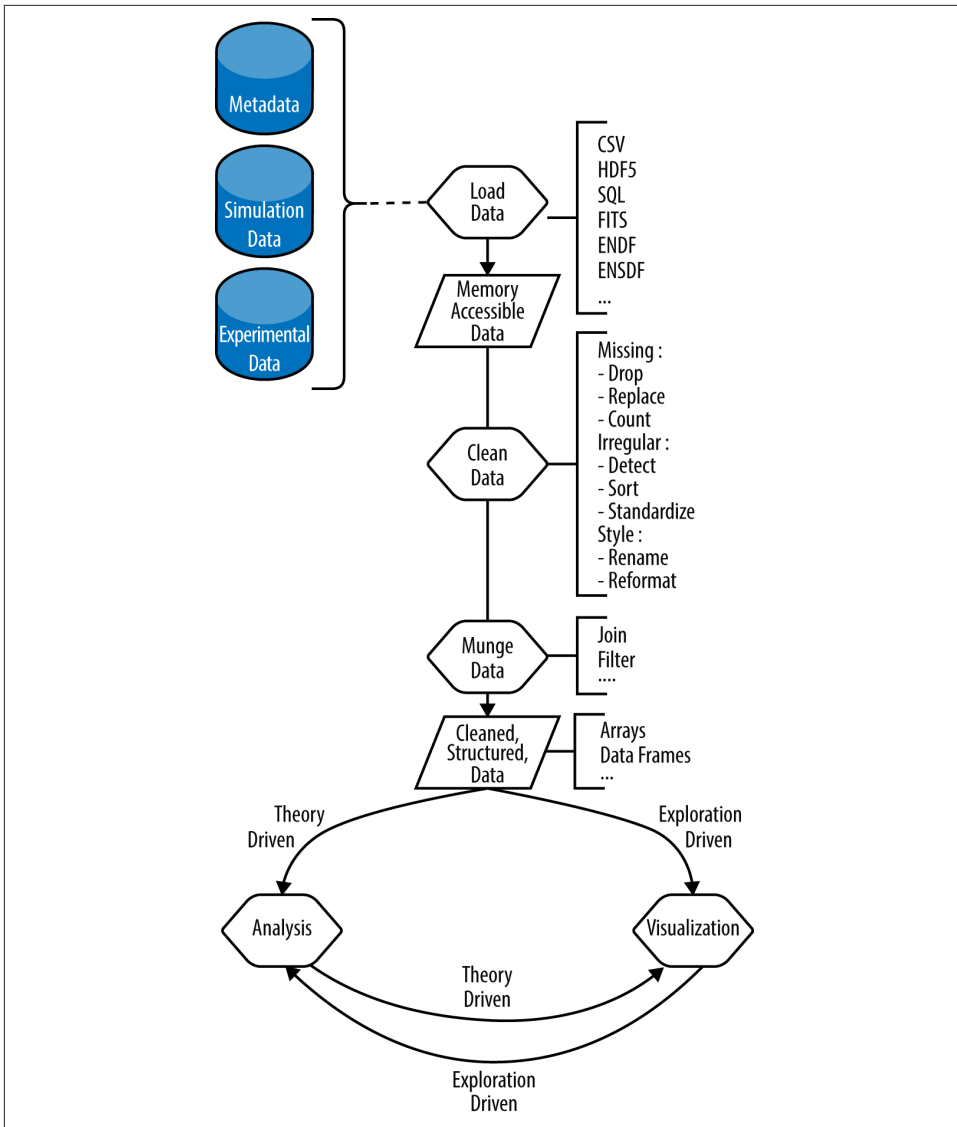


Figure 7-1. Data preparation for analysis and visualization

Due to this involved sequence of necessary tasks, many scientists spend vast portions of their research careers:

- Cleaning data by hand
- Executing analysis steps one at a time
- Using a mouse when creating plots

- Repeating the whole effort when new data appears or the process must be tweaked

However, more efficient scientists automate these processes and have more time for research. These efficient scientists spend their research careers doing the following:

- Scripting their data pipeline
- Gaining confidence in their results
- Doing additional research
- Publishing more papers

These scientists must invest extra effort up front, but they benefit from increased efficiency later. By taking the time to generate scripts that automate their pipelines (data cleaning, processing, analysis, plotting, etc.), efficient scientists can more rapidly incorporate new data into their analyses, test new analysis steps, and tweak plots. The pipeline can be simple or complex and may include a wide array of actions, such as:

- Data entry
- Data cleanup
- Building any necessary software
- Running simulation analyses
- Data post-processing
- Uncertainty estimation
- Generating tables and plots for publication
- Building papers around that work

An astute reader may note that the former kind of scientist may publish results faster if the datasets are pristine and reviewers are generous. Fools and optimists are invited to rely on these two miracles. Realists should automate. Furthermore, those who are sickened by the rampant lack of reproducibility in scientific computation should move to automate their pipelines, irrespective of the cost-benefit ratio. Fully scripted analysis and visualization is a necessary feature of reproducible science. Though incentive structures in the research sciences currently fall short of rewarding reproducibility, the tables are turning. Be on the righteous side of history—heed the words of Mario Savio, and automate your methods:

There's a time when the operation of the machine becomes so odious, makes you so sick at heart, that you can't take part! You can't even passively take part! And you've got to put your bodies upon the gears and upon the wheels... upon the levers, upon all the apparatus, and you've got to make it stop! And you've got to indicate to the people who

run it, to the people who own it, that unless you're free, the machine will be prevented from working at all!

When you're automating your methods, the first thing to automate is the processing of raw data. How the data processing step is performed varies greatly depending on whether your work is based on experimentation or simulation. The next sections will cover each of those cases, as well as the metadata that is associated with them.

Experimental Data

Experimental data presents unique analysis challenges to the scientist. Such data usually takes the form of detector responses and physical measurements resulting from various experiments in the physical sciences.

Experiments that observe a variable changing over time will produce time series data, where the independent variable is time and the dependent variable is the observation. Time series data such as this is often stored in flat tabular plain-text or simple CSV files. An example might be an experiment that seeks to identify the decay constant of an isotope based on its dwindling radiation signature over time. Such data might be structured as in [Table 7-1](#).

Table 7-1. Isotope decay data (decays.csv)

Time(s)	Decays (#)
0	10.0
1	1.353352832
2	0.183156389
3	0.024787522
4	0.003354626
5	0.000453999
6	6.1442e-05
7	8.315e-06
8	1.126e-06
9	1.52e-07
...	...

In its raw form, this data may be stored in a comma-separated or otherwise delimited plain-text format, as seen here:

```
# Time (s), Decays (#)
0,10.0
1,1.353352832
2,0.183156389
3,0.024787522
4,0.003354626
5,0.000453999
6,6.1442e-05
7,8.315e-06
8,1.126e-06
9,1.52e-07
```

Experiments that seek certain parametric relationships between variables, however, may produce multidimensional, structured, or tabular data. Many experiments have qualities of both and produce multidimensional, structured time series data. The possibilities are endless, but the structure of data often dictates the format it must be stored in. While time series can be stored easily in CSV format, very complex structured data typically calls for a standardized database format.

In special cases, scientific communities adopt their own very domain-specific file formats for storing experimental data. Astrophysicists, for example, store their enormous libraries of telescope images and telescope calibration metadata together in the specialized Flexible Image Transport System (FITS) file format. In nuclear physics, on the other hand, physicists do not deal with images. Rather, they deal primarily with particle interaction probabilities called *cross sections*. Many international standards exist for storing this type of data, but the most common nuclear data format is the Evaluated Nuclear Data File (ENDF) format.

Once formatted, evaluated data like the data that appears in ENDF or FITS formats is ready to be used in simulations.

Simulation Data

Simulations are just experiments *in silico*. A wise man, Professor Paul P.H. Wilson, used to tell his students and colleagues that scientific computation is just experimentation for scientists who like to control more variables. An experimentalist, he would explain, begins with the whole world of uncontrolled variables and designs an experiment that carefully controls those variables one at a time, step by step, until only the experimental variables are left. The computationalist, on the other hand, starts with a completely controlled simulation and carefully, one at a time, *releases* variables.

Because of the higher degree of control in simulation, simulation data output formats are often already quite clean and well controlled. Raw simulation data typically resides in databases.

For provenance, databases may need to be accompanied by data about the simulation, such as:

- The date the simulation was run
- The names and contents of the input files
- The version numbers of libraries used

This type of data, present in both experiments and simulations, is called *metadata*.

Metadata

Metadata is data about data. It is important to include metadata because the results produced by an experiment or a simulation sometimes fail to capture all of its features.

In an experiment, detector parameters, the date of the experiment, background radiation readings from another detector, and more, can all affect the interpretation of results, but these factors may not have been captured in the experimental data output stream. They can instead be captured in the metadata.

Metadata is not limited to experimentation, however. Metadata that may affect the interpretation of the results of a simulation include simulation ID numbers, library dependency version numbers, input file content, and more.

For reproducibility, as much of this data should be included in the workflow as possible. From a data processing perspective, this metadata also often needs to be joined with the experimental or simulation database. The steps necessary for preparing metadata for this process will vary from application to application. However, metadata should be held to the same standards of formatting and reproducibility as simulation or experimental data.

Of course, once all of the experimental, simulation, and/or metadata has been prepared, it must be loaded into a form that can be analyzed. In [Figure 7-1](#), this next step is the “Load Data” step. It will be covered in the following section.

Loading Data

Many packages in Python enable data to be loaded quickly into a memory-accessible data structure ready for cleaning, munging, analysis, and visualization. More about data structures will be covered in [Chapter 11](#). The choice of appropriate data structure depends profoundly on the size and type of the data as well as its analysis and visualization destiny. This section will merely cover how to load data into various analysis-ready forms using four tools in the Python ecosystem:

- NumPy

- PyTables
- Pandas
- Blaze

When you are choosing among tools like these, a number of factors come into play.

Size is the first parameter to consider. For big, dense arrays of numbers or for enormous suites of images, loading all of the data into memory at once is not recommended. Rather, loading *chunks* of the data into memory while cleaning, processing, or exploring it might be advised. For this and similar out-of-core computations on data exceeding the bounds of system memory, a database choice that emphasizes these features is warranted. On that topic, later sections will address loading data into PyTables and Blaze.

The type of the data also may determine the appropriate data structure. Structured data in a relational database format may be easiest to handle using extraction and exploration tools in the `sqlite3` or `pandas` packages.

All of that said, some data is small enough to fit into memory all at once, and much of the data that physicists encounter or produce is best represented as an array of numbers. For this application, the right tool is the numerical Python package, `numpy`.

NumPy

Due to their power, many solutions involve NumPy arrays. NumPy arrays will be covered in detail in [Chapter 9](#). For now, simply know that NumPy arrays are data structures for holding numbers. One easy way to transform a file into a NumPy array is with the `loadtxt` function. Using this function, plain-text files holding columns of text delimited by commas, spaces, or tabs can be loaded directly into a NumPy array. The decay data in our earlier CSV-formatted example can be loaded directly into a NumPy array shown here:

```
import numpy as np ❶
decays_arr = np.loadtxt('decays.csv', delimiter=",", skiprows=1) ❷
```

❶ Import `numpy` with the alias `np`.

❷ Create a NumPy array object called `decays_arr` using the `loadtxt()` function.

In this example, the `numpy` package is loaded and given the short alias `np`. Next, a variable called `decays_arr` is declared and set equal to the output of the `loadtxt()` function. The variable `decays_arr` is a NumPy array. In this case, the `loadtxt()` function is parameterized with only one mandatory variable, the filename. The two optional variables are the delimiter (a comma), and the number of rows to skip (the header row, which is not numbers). Though not all were used in this example, many other

options are available to customize the way a file is read with `loadtxt()`. To learn more about those, consult the `numpy.loadtxt()` documentation using the `help` command in IPython:

```
In [1]: import numpy as np ❶
```

```
In [2]: help(np.loadtxt) ❷
```

- ❶ Import numpy with the alias `np`.
- ❷ Learn more about the `loadtxt()` function.

Once data has been loaded into a NumPy array, one of the fastest ways to deal with that data for large-scale problems is to enlist the help of PyTables.

PyTables

As you will learn in [Chapter 10](#), PyTables provides many tools for converting HDF5 data into analysis-ready NumPy arrays. Indeed, because PyTables can help load, modify, and manipulate HDF5 data in the form of NumPy arrays, it is a strong motivator for the use of HDF5 as a raw data format. Perhaps the decay data in the previous example could be more easily manipulated in the future if it were stored in an HDF5 database—PyTables can help with that. Any data loaded into a NumPy array can be quickly and easily saved as an HDF5 database. So, once data has been loaded as a NumPy array, it is ready for use with PyTables. This allows for faster filtering, joins, and analysis later. However, PyTables and HDF5 are most useful for storing and manipulating dense arrays of numbers, so if your data is heterogeneous or sparse, or contains structured, relational information, it may be best stored in another format. If that is the case, a multifunction Python package like `pandas` may be more appropriate than PyTables. For information on when and how to load data into Pandas, read on.

Pandas

Pandas is currently the easiest to use and most broadly applicable of all of the data analysis tools in the Python ecosystem. It is a good alternative to the previously discussed tools, especially if your data is not in a format that is supported by NumPy (CSV or plain text) or PyTables (HDF5). Also, Pandas may be the right choice even for those formats if your data is not arrays of numbers or is the kind of data that you would like to view, filter, browse, and interact with.

Pandas cleanly handles reading and writing of many of the data formats encountered by scientists in the wild:

- CSV
- Excel

- HDF
- SQL
- JSON
- HTML
- Stata
- Clipboard
- Pickle

Also, loading data into a Pandas format is very straightforward. Note that the capability of `numpy.loadtxt()` can be repeated in Pandas with very similar syntax:

```
import pandas as pd ❶
decays_df = pd.read_csv('decays.csv') ❷
```

- ❶ Import the pandas package and alias it as pd.
- ❷ Create a data frame object that holds the data loaded by the `read_csv()` function.

A lovely quality in Pandas is that once data has been loaded, it can be converted into any of the other supported formats. To write this data to an HDF5 file, we need to add just one line to the previous example:

```
import pandas as pd ❶
decays_df = pd.read_csv('decays.csv') ❷
decays_df.to_hdf('decays.h5', 'experimental') ❸
```

- ❶ Import the pandas package and alias it as pd.
- ❷ Create a data frame object that holds the data loaded by `read_csv()`.
- ❸ Convert it to HDF5, giving it the filename `decays.h5`, and create a group node called “experimental” where this data will be stored.

Pandas is a top-notch tool for data analysis with Python. To learn how to fully wield the power of Pandas, refer to *Python for Data Analysis* by the lead developer of Pandas, Wes McKinney (O’Reilly). The data analysis in that book goes way beyond the scope of this section. Here, we simply mean to introduce the existence of this tool, alongside a few other tools that might also be considered. The final such data analysis tool that we will introduce is Blaze. Like Pandas, it is intended for easily loading data into an analysis-ready format and emphasizes ease of conversion between formats.

Blaze

Blaze is another Python tool capable of converting data from format to format. This tool is still in active development, but possesses impressive capabilities already. In Blaze, the CSV data might be dealt with as a Blaze data descriptor or as a Blaze Table object. The following example shows the transformation from CSV to data descriptor, and an additional transformation from data descriptor to Blaze Table object:

```
import blaze as bz ❶  
csv_data = bz.CSV('decays.csv') ❷  
decays_tb = bz.Table(csv_data) ❸
```

- ❶ The `blaze` package is imported and given the alias `bz`.
- ❷ Next, the CSV data is transformed into Blaze data with the `CSV()` constructor.
- ❸ Finally, that data descriptor, `csv_data`, is transformed into a Blaze Table.

This example illustrates how one type of Blaze object can be quickly converted to another data structure within Blaze quite straightforwardly. Since the aim of Blaze is to support conversion between many data formats (or “backends,” in Blaze-speak), it may be the right tool to use if your data files must be converted from one memory-accessible format to another.



Blaze is still under active development. Unlike the other tools discussed here (NumPy and PyTables in particular), it is not yet fully stable. However, the features discussed here are quite mature, and it will be a tool to watch closely as it improves.

This flexibility is likely to make Blaze very handy for certain applications, as this tool not only provides an interface for converting between many data formats (CSV, HDF5, SQLite, etc.) but also provides an interface to support workflows using many computational engines. Blaze uses symbolic expression and typing systems to communicate with other tools including Pandas, NumPy, SQL, Mongo, Spark, and PyTables. Access to computational engines like those, which are capable of manipulating the data, is essential for the next step in the process of data analysis: *cleaning and munging*.

Cleaning and Munging Data

Data *munging* (or *wrangling*) is a term used to mean many different things within the broad scope of *dealing with data*. Typically, as in [Figure 7-1](#), the term refers to the process of converting data from a raw form to a more well-structured form appropriate for plotting and mathematical transformation.

The scientist may *wrangle* the data by grouping, filtering, aggregating, collapsing, or expanding it. Depending on your particular data, this step may need to happen before the data is *cleaned*, or may not have to happen until after. Cleaning data can also take many forms. Typically, this task deals with imperfect, incomplete, and disorganized data.

Of course, ideally, experimentalists in the physical sciences use sophisticated, automated, comprehensive data acquisition systems that produce clean, flawless datasets in intuitive formats. However, even such systems can produce imperfect data in extreme conditions.

The decay data being used in the previous examples, for instance, might have errors if other radioactivity were present in the laboratory. Additionally, if the power to the detector were cut off in an electrical blackout, data would be unavailable for a period of time.

To explore this, let's consider a more realistic version of the data we dealt with before. It may have machine-generated timestamps instead of integer numbers of seconds, and it may have missing or imperfect data. Imagine, for example, that about 15 seconds into the experiment, a colleague walked through the room carrying a slightly more stable radioactive source, emitting two decays per second. Additionally, imagine that a few seconds later, the lights in the room flashed off for a few seconds—the storm outside must have interrupted power to the lab. The resulting data stream looks like this:

```
#Time,Decays
```

```
2014-11-08T05:19:31.561782,10.0
2014-11-08T05:19:32.561782,1.35335283237
2014-11-08T05:19:33.561782,0.183156388887
2014-11-08T05:19:34.561782,0.0247875217667
2014-11-08T05:19:35.561782,0.00335462627903
2014-11-08T05:19:36.561782,0.000453999297625
2014-11-08T05:19:37.561782,6.14421235333e-05
2014-11-08T05:19:38.561782,8.31528719104e-06
2014-11-08T05:19:39.561782,1.12535174719e-06
2014-11-08T05:19:40.561782,1.52299797447e-07
2014-11-08T05:19:41.561782,2.06115362244e-08
2014-11-08T05:19:42.561782,2.78946809287e-09
2014-11-08T05:19:43.561782,3.77513454428e-10
2014-11-08T05:19:44.561782,5.10908902806e-11
2014-11-08T05:19:45.561782,6.91440010694e-12
2014-11-08T05:19:46.561782,9.35762296884e-13
2014-11-08T05:19:47.561782,2.0000000000000000 ❶
2014-11-08T05:19:48.561782,2.0000000000000000
2014-11-08T05:19:49.561782,2.0000000000000000
2014-11-08T05:19:50.561782,2.0000000000000000
2014-11-08T05:19:51.561782,2.0000000000000000
2014-11-08T05:19:52.561782,2.0000000000000000
2014-11-08T05:19:53.561782,2.0000000000000000
```

```

2014-11-08T05:19:54.561782,2.0000000000000000
2014-11-08T05:19:55.561782,2.0000000000000000
2014-11-08T05:19:56.561782,1.92874984796e-21
2014-11-08T05:19:57.561782,2.61027906967e-22
2014-11-08T05:19:58.561782,3.5326285722e-23
2014-11-08T05:19:59.561782,4.78089288389e-24
2014-11-08T05:20:00.561782,6.47023492565e-25
2014-11-08T05:20:01.561782,8.7565107627e-26
2014-11-08T05:20:02.561782,1.18506486423e-26
2014-11-08T05:20:03.561782,1.60381089055e-27
2014-11-08T05:20:04.561782,2.1705220113e-28
2014-11-08T05:20:05.561782,2.93748211171e-29
2014-11-08T05:20:06.561782,3.97544973591e-30
2014-11-08T05:20:07.561782,5.38018616002e-31
2014-11-08T05:20:08.561782,7.28129017832e-32
2014-11-08T05:20:09.561782,9.85415468611e-33
2014-11-08T05:20:10.561782,1.3336148155e-33
2014-11-08T05:20:11.561782,1.80485138785e-34
2014-11-08T05:20:12.561782,NaN ❷
2014-11-08T05:20:13.561782,NaN
2014-11-08T05:20:14.561782,NaN
2014-11-08T05:20:15.561782,NaN
2014-11-08T05:20:16.561782,8.19401262399e-39
2014-11-08T05:20:17.561782,1.10893901931e-39
2014-11-08T05:20:18.561782,1.50078576271e-40
2014-11-08T05:20:19.561782,2.03109266273e-41
2014-11-08T05:20:20.561782,2.74878500791e-42

```

- ❶ Uh oh, it looks like the reading was overwhelmed by another source moving through the room.
- ❷ At this point, it seems the detector was off, and no readings were made.



NaN entries, as in this example, indicate that no number is stored in memory at the place where the data should be. NaN stands for “Not a Number.”

Some experimentalists might see the NaN entries and immediately assume this data must be thrown away entirely. However, since experiments are often expensive and time-consuming to conduct, losing an entire run of data due to minor blips like this is often unacceptable. Concerns about data quality and inconsistencies are very common in science. Sometimes, dates are listed in a dozen different formats. Names are inconsistent across files. And sometimes data is erroneous. In this case, the section with too-high (2.0) counts due to the external radioactive source dwarfing the actual signal must be dealt with. How this section of the data is handled is a choice for the

experimenter. Whatever the choice, however, tools exist to assist in the implementation.

The data from this run is ugly, but can it be saved with intelligent cleaning and modern tools? The following section will discuss one way to deal with missing data using Pandas.

Missing Data

Sometimes, data is missing. In some situations, a missing data point may be appropriate or expected, and can be handled gracefully. Often, however, it may need to be replaced with a default value, its effect on the statistical analysis of the results may need to be taken into consideration, or those data points may just need to be dropped.

Pandas is especially helpful in the event of missing data. In particular, Pandas has special methods for identifying, dropping, and replacing missing data.

With only a few lines in IPython, the NaN rows from the previous data can be dropped from the dataset entirely:

```
In [1]: import pandas as pd

In [2]: decay_df = pd.read_csv("many_decays.csv")

In [3]: decay_df.count() ❶
Out[3]:
Time      50
Decays    46
dtype: int64

In [4]: decay_df.dropna() ❷
Out[4]:
```

	Time	Decays
0	2014-11-08T05:19:31.561782	1.000000e+01
1	2014-11-08T05:19:32.561782	1.353353e+00
2	2014-11-08T05:19:33.561782	1.831564e-01
3	2014-11-08T05:19:34.561782	2.478752e-02
4	2014-11-08T05:19:35.561782	3.354626e-03
5	2014-11-08T05:19:36.561782	4.539993e-04
6	2014-11-08T05:19:37.561782	6.144212e-05
7	2014-11-08T05:19:38.561782	8.315287e-06
8	2014-11-08T05:19:39.561782	1.125352e-06
9	2014-11-08T05:19:40.561782	1.522998e-07
10	2014-11-08T05:19:41.561782	2.061154e-08
11	2014-11-08T05:19:42.561782	2.789468e-09
12	2014-11-08T05:19:43.561782	3.775135e-10
13	2014-11-08T05:19:44.561782	5.109089e-11
14	2014-11-08T05:19:45.561782	6.914400e-12
15	2014-11-08T05:19:46.561782	9.357623e-13

```

16 2014-11-08T05:19:47.561782 2.000000e+00
17 2014-11-08T05:19:48.561782 2.000000e+00
18 2014-11-08T05:19:49.561782 2.000000e+00
19 2014-11-08T05:19:50.561782 2.000000e+00
20 2014-11-08T05:19:51.561782 2.000000e+00
21 2014-11-08T05:19:52.561782 2.000000e+00
22 2014-11-08T05:19:53.561782 2.000000e+00
23 2014-11-08T05:19:54.561782 2.000000e+00
24 2014-11-08T05:19:55.561782 2.000000e+00
25 2014-11-08T05:19:56.561782 1.928750e-21
26 2014-11-08T05:19:57.561782 2.610279e-22
27 2014-11-08T05:19:58.561782 3.532629e-23
28 2014-11-08T05:19:59.561782 4.780893e-24
29 2014-11-08T05:20:00.561782 6.470235e-25
30 2014-11-08T05:20:01.561782 8.756511e-26
31 2014-11-08T05:20:02.561782 1.185065e-26
32 2014-11-08T05:20:03.561782 1.603811e-27
33 2014-11-08T05:20:04.561782 2.170522e-28
34 2014-11-08T05:20:05.561782 2.937482e-29
35 2014-11-08T05:20:06.561782 3.975450e-30
36 2014-11-08T05:20:07.561782 5.380186e-31
37 2014-11-08T05:20:08.561782 7.281290e-32
38 2014-11-08T05:20:09.561782 9.854155e-33
39 2014-11-08T05:20:10.561782 1.333615e-33
40 2014-11-08T05:20:11.561782 1.804851e-34 ③
45 2014-11-08T05:20:16.561782 8.194013e-39
46 2014-11-08T05:20:17.561782 1.108939e-39
47 2014-11-08T05:20:18.561782 1.500786e-40
48 2014-11-08T05:20:19.561782 2.031093e-41
49 2014-11-08T05:20:20.561782 2.748785e-42

```

- ① The data frame method `count()` successfully ignores the NaN rows automatically.
- ② The `dropna()` method returns the data excluding all rows containing a NaN value.
- ③ Here, the time skips ahead 5 seconds, past the now-missing NaN rows.

Now the data is much cleaner, as the offending missing data has been dropped entirely. This automation of dropping NaN data is quite useful when you're preparing data for the next step: analysis.

Analysis

A fleet of tools is available for loading, processing, storing, and analyzing data computationally. In a Python data analysis environment, the **numpy**, **scipy**, and **pandas** packages are the big hammers for numerical analysis. However, many packages within the SciPy and SciKits ecosystems complement those hard-working tools. Some Python-based analysis toolkits to use, organized by discipline, can be found on the

SciPy and SciKits websites. There are too many to list here. However, some highlights include:

- Astronomy and astrophysics
 - **Astropy**: Core astronomy and astrophysics tools; includes FITS, ASCII, VOTable, and XML file format interfaces
 - **PyRAF**: Python-based interface to IRAF
 - **SpacePy**: Data, analysis, and plotting tools for space sciences
 - **SunPy**: Solar data analysis environment
- Geophysics
 - **OSGeo**: GIS data import/export and analysis
 - **Basemap**: 2D mapping
- Engineering
 - **PyNE**: Toolkit for nuclear engineering
 - **scikit-aero**: Aeronautical engineering calculations in Python
- Mathematics
 - **SymPy**: Symbolic mathematics in Python
- Neuroscience
 - **NIPY**: Neuroimaging in Python
- Quantum physics and chemistry
 - **QuTiP**: Quantum Toolbox in Python, simulating dynamics of open quantum systems
 - **PyQuante**: Python for Quantum Chemistry

The analysis step is very application specific and requires domain knowledge on the part of the physicist. A large part of analysis in the physical sciences, when models are derived, confirmed, or disproved based on experimental data, can be termed *inference* or *abstraction*. Abstraction can be an art as well as a science. It can be driven by, generally speaking, either side of the equation: the model or the data.

Model-Driven Analysis

In the case of the decay data, the model-driven analysis is very simple. To determine the decay constant of the isotope in question, we can fit an exponential to the data. The well-known and accepted model for the decay of a radioactive isotope is $N = N_0 e^{-\lambda t}$.

Of course, that is a simple example. Most analysis in the physical sciences requires many steps of filtering and merging of data as well as integrations, interpolations, scaling, and so on.

A Note on Floating-Point Arithmetic

An excellent resource as you embark on the task of implementing your own numerical analysis algorithms is David Goldberg's paper, [What Every Computer Scientist Should Know About Floating-Point Arithmetic](#). It sounds dry, but it truly is essential reading for any researcher who deals with algorithms that manipulate floating-point numbers.

The paper contains a series of best practices for reducing numerical error obtained during even simple operations such as summation, multiplication, and division.

As an example, the accuracy of simply summing a list of floating-point numbers varies strongly according to the order in which those numbers are summed. Errors, in general, are reduced when smaller numbers are added before larger ones, but more complex algorithms such as the Kahan summation algorithm improve greatly upon simple ordering.

While many of those techniques have now been encapsulated in numerical libraries, some algorithms for data analysis in physics have yet to be written. Furthermore, having a working knowledge of the implementation of the algorithms in the libraries being used for your analysis will help you to determine the applicability of an algorithm to your problem, or to differentiate two options when multiple algorithmic methods are available for a single problem.

Numerical Recipes: The Art of Scientific Computing, by William Press et al., is an excellent reference when implementing a common algorithm is necessary, and was once required reading for computational analysis. It is particularly relevant for model-driven analysis in the physical sciences, which often requires various methods for numerical integrations, solutions of differential equations, and evaluation of large systems of equations. This tome illuminates useful algorithms covering such elements of numerical analysis as:

- Interpolation and extrapolation
- Integration and derivation
- Evaluation of functions
- Inverse functions
- Fermi-Dirac integrals
- Random numbers

- Sorting
- Root finding

Beyond these algorithms, more elaborate methods exist. Many modern algorithms emphasize analysis that does not seek to measure a model based on data. Instead, it often seeks to *generate* models based on data. This is often termed *data-driven* analysis.

Data-Driven Analysis

In data-driven analysis, fancier methods are common. These include clustering algorithms, machine learning processes, and exotic statistical methods. Such algorithms are increasingly available in standard open source libraries and are increasingly common in physical sciences applications. They typically are used to infer models from the data (rather than confirm or deny models using the data). The Python ecosystem possesses many tools enabling such algorithms, including:

- Machine learning and clustering
 - `scikit-learn`
 - `PyBrain`
 - `Monte`
 - `PyPR`
 - `scipy-cluster`
- Statistics
 - `Statsmodels`
 - `PyBayes`

Which algorithms and implementations of those algorithms to use and how to apply them to your data will be choices that are driven by your science. Whatever tools or techniques you use, however, data analysis results in conclusions that can, usually, be visualized.

Visualization

How you visualize your data is the first thing anyone will notice about your paper, and the last thing they'll forget. For this reason, visualization should be taken very seriously and should be regarded as a first-class element of any data analysis workflow.

A lot has been learned about how to present data and information most effectively. Much of this knowledge has emerged from business and marketing contexts.

In science—unlike in business, perhaps—visualization must not attempt to convince or persuade. Bias, obfuscation, and distortion are the mortal enemies of scientific visualization. Visualization in science must demonstrate, clarify, and explain.

Indeed, visualization best practices share many qualities with Python best practices. Python contains an easter egg: a poem on Python best practices, “The Zen of Python,” by Tim Peters, is printed in response to the command `import this`. Though it was intended to illuminate guidelines for good Python programming, its first few lines also capture key rules that can be equally well applied to the display of information:

Code	Output
<code>import this</code>	<p>The Zen of Python, by Tim Peters</p> <p>Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex. Complex is better than complicated. Flat is better than nested. Sparse is better than dense. Readability counts. Special cases aren't special enough to <code>break</code> the rules. Although practicality beats purity.</p>

Combining these notions with insights gleaned from Edward Tufte’s book *The Visual Display of Quantitative Information* and from Matthew Terry, physicist-turned-software-engineer, I hereby recommend the following “Zen of Scientific Visualization”:

- Beautiful is better than ugly.
- Simple is better than complex.
- Complex is better than complicated.
- Clear is better than dense.
- Communicating is better than convincing.
- Text must not distract.
- People are not mantis shrimp.

The final recommendation may seem a bit odd. However, consider the **biology of the mantis shrimp**. With vastly more color-perceiving cones in its eyes than a human, the mantis shrimp is able to distinguish colors with vastly greater resolution. People are not mantis shrimp. They are often confused and distracted by too many colors on a

single plot. Additionally, many people in the physical sciences are colorblind,¹ so plots that rely too heavily on color to communicate may be somewhat discriminatory. Avoid complicated colormaps, and be sensitive to color blindness by avoiding heavy reliance on color.

Visualization Tools

Many libraries for plotting data exist. This section will introduce a few of the key libraries available for plotting publication-quality scientific data. The tools covered include:

Gnuplot

Best for simple plots, as the syntax is peculiar

matplotlib

A powerful plotting library in Python, robust and widely used

Bokeh

Produces interactive plots appropriate for the Web, also interfaces with matplotlib

Inkscape

Good for hand editing of scalable vector graphics

This section will introduce these tools by demonstrating their use to plot the decay data from our previous examples in a simple line plot. This introduction should be enough to help you get started with the right tool very quickly when you need to plot you work.

These tools are all available within the scientific Python ecosystem, with one exception: gnuplot. Gnuplot is not a Python tool, but it stands strong as a plotting option nonetheless.

Gnuplot

Gnuplot is a key tool in the physics arsenal. Though it is not a Python tool, it is sufficiently embedded in the physics community that we would be remiss were we to fail to address it here. While it has never had the most beautiful graphics, plentiful error messages, or pleasing syntax, physicists have loved it since its birth. Gnuplot is a workhorse in physics, for better or worse.

¹ While efforts are being made to improve the situation, it is an unfortunate fact that there is a long-standing gender imbalance in the physical sciences: even today, there are far more males than females in these fields. Because color vision deficiencies are more common in males than females, this imbalance means that color blindness is, in turn, more common in the physical sciences than in the general population.

The gnuplot interpreter is launched with the command `gnuplot`. In that interpreter, it is possible to enter commands to construct a plot. However, the most reproducible way to use gnuplot is by creating a script, which is then provided as an argument to the `gnuplot` command.



Exercise: Learn More About Gnuplot

Since gnuplot is a command-line tool, it is possible to learn more about it using the `man` command.

1. Open a terminal.
2. Find out how to use the `gnuplot` command via its `man` page.

(Hint: For more on `man` pages, see [Chapter 1](#).)

The following gnuplot script can be used to plot the decay data, along with a title and axis labels:

```
set title 'Decays' ❶
set ylabel 'Decays '
set xlabel 'Time (s)'
set grid ❷
set term svg ❸
set output 'plot_gnuplot.svg' ❹
plot 'decays.csv' every ::1 using 1:2 with lines ❺❻❼
```

- ❶ The `set` keyword defines variables like the title and axis labels.
- ❷ `set` can also add predefined customizations—in this case, grid lines.
- ❸ This sets the output terminal (file) to the SVG file format.
- ❹ This names the output file.
- ❺ The `plot` command accepts data from the input file.
- ❻ Of the rows in the input file, print all except the first.
- ❼ Of the columns in the input file, plot 1 and 2 against one another.

This script can be run with the `gnuplot` command on the command line. Try placing this code in a file called `decay_plot.gnuplot` and running the command `gnuplot decay_plot.gnuplot`. This script produces the visualization in [Figure 7-2](#).

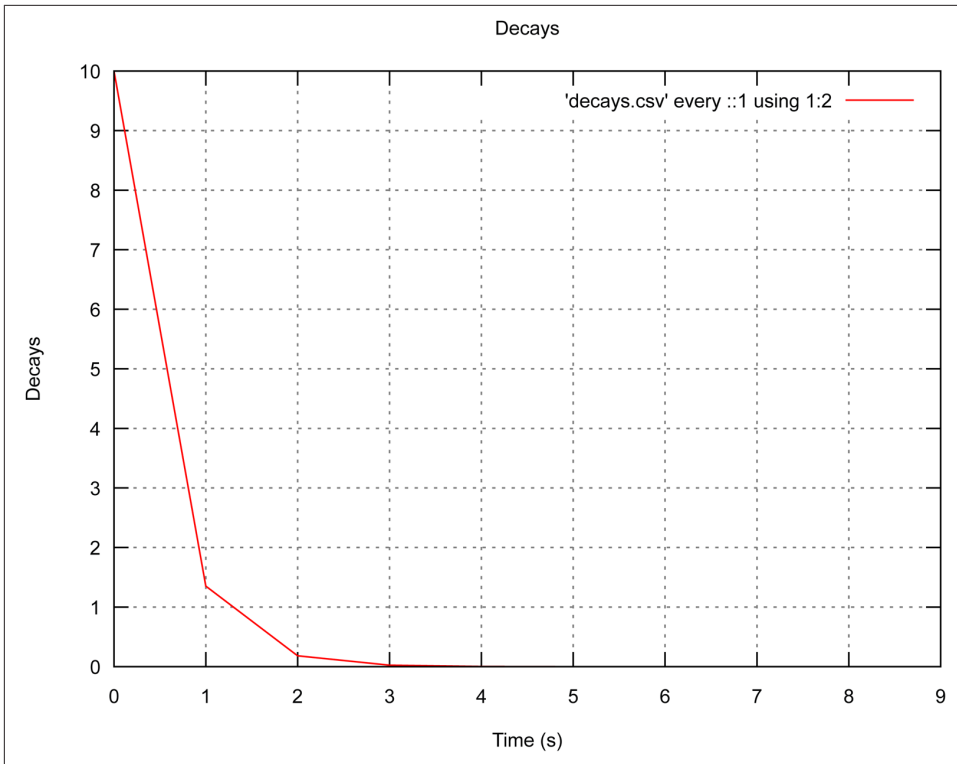


Figure 7-2. Gnuplot plot

By default, gnuplot uses red as the first color to plot in a line plot. Thus, the resulting plot line is red, though we did not dictate a line color. While this is handy, the second deployed default line color in gnuplot is green, which is unfortunate for color-blind people.



Exercise: Change the Color

1. Open a terminal window.
2. Create the *decay_plot.gnuplot* file and [Figure 7-2](#).
3. Modify the script to produce a blue line rather than red.

(Hint: Use the `man` page and documentation online to determine how to change the color of the plot line.)

Though gnuplot is very simple and may be the first plotting library taught to a physicist on the command line, more effective physics can be accomplished if the plotting library is able to make direct use of the data preparation steps described in the previ-

ous sections. Additionally, graphics features implemented in more modern Python packages are somewhat superior aesthetically to the graphical capabilities in gnuplot. One such alternative is matplotlib.

matplotlib

The workhorse for scientific plotting in Python is **matplotlib**. We can reproduce our gnuplot plot with matplotlib by running the following Python script to create the new file:

```
import numpy as np ❶

# as in the previous example, load decays.csv into a NumPy array
decaydata = np.loadtxt('decays.csv', delimiter=",", skiprows=1)

# provide handles for the x and y columns
time = decaydata[:,0]
decays = decaydata[:,1]

# import the matplotlib plotting functionality
import pylab as plt

plt.plot(time, decays) ❷

plt.xlabel('Time (s)')
plt.ylabel('Decays')
plt.title('Decays')
plt.grid(True) ❸
plt.savefig("decays_matplotlib.png") ❹
```

- ❶ First we import numpy, so that we can load the data.
- ❷ This generates a plot of decays vs. time.
- ❸ This adds gridlines.
- ❹ This saves the figure as a PNG file (matplotlib guesses based on the extension).

This Python script can be run with the python command on the command line. To create the script on your own, place the code into a file called *decay_plot.py*. Running the command `python decay_plot.py` produces the plot in **Figure 7-3**.

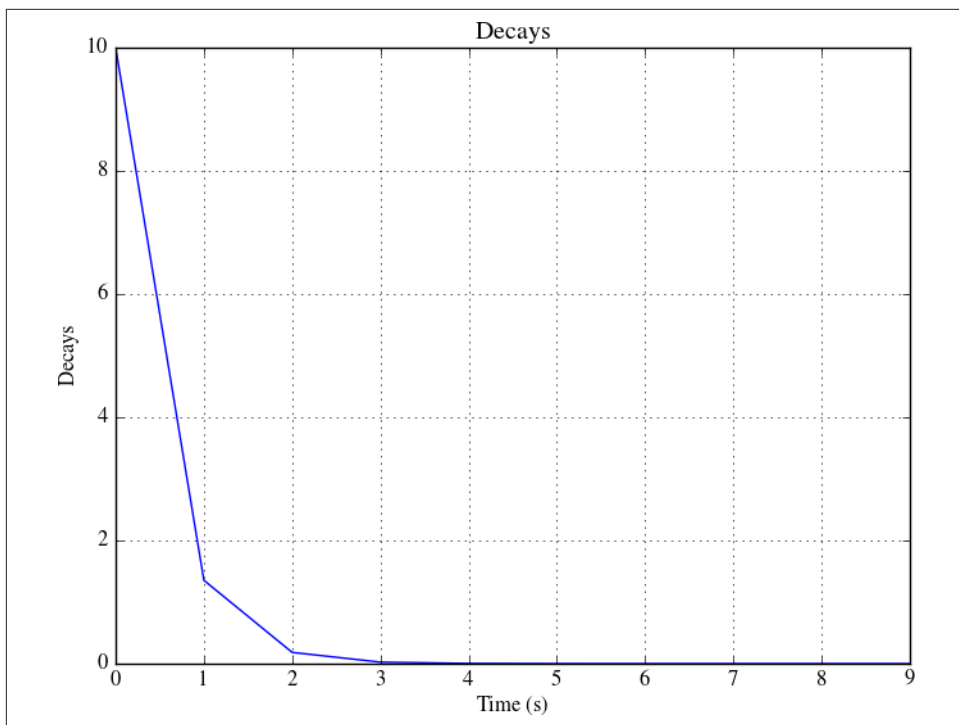


Figure 7-3. *matplotlib* plot

This plot is not very exciting, of course. When data is more complex—perhaps 2D or 3D, with multiple dependent parameters—*matplotlib* begins to show its true (very powerful) colors. Equipped with a plethora of plot types and aesthetic options, the power under the hood in *matplotlib* can be almost overwhelming. In such a situation, the *gallery* in *matplotlib* comes to the rescue.

The gallery

The best way to start with *matplotlib* is with the [gallery](#). Nearly every feature of the *matplotlib* plotting library is demonstrated by an example plot in the gallery, along with the source code that generated it. It contains a wealth of example scripts, lovingly created by the developers and users of the library.

In a physics context, the gallery is a powerful resource due to the speed with which it enables a researcher to identify desired features of a plot. With the source code for many features available, you can mix and match them to produce a compelling figure with your own scientific data, or, really, any customization at all.

Indeed, *matplotlib* plots can be used for nearly any purpose. One of the coolest examples in the gallery is certainly the polar plot used in the *matplotlib* logo ([Figure 7-4](#)).



Figure 7-4. matplotlib logo

In 2010, one of the authors of this book had the opportunity to help organize a talk by the creator of this extraordinary library, John D. Hunter.

When someone like this comes to give a talk about their plotting tool, the pressure is on: one must make an excellent flyer to advertise the event. The first step in making a flyer for the talk was to customize the script for the cool polar plot from the gallery. With matplotlib annotations, text boxes were added at specific x , y coordinates. To announce the event at the University of Wisconsin, both the Python script shown here and the resulting PDF plot [Figure 7-5](#) were emailed to the students and staff:

```
#!/usr/bin/env python ❶

# Import various necessary Python and matplotlib packages
import numpy as np
import matplotlib.cm as cm ❷
from matplotlib.pyplot import figure, show, rc ❸
from matplotlib.patches import Ellipse ❹

# Create a square figure on which to place the plot
fig = figure(figsize=(8,8))

# Create square axes to hold the circular polar plot
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], polar=True)

# Generate 20 colored, angular wedges for the polar plot
N = 20
theta = np.arange(0.0, 2*np.pi, 2*np.pi/N)
radii = 10*np.random.rand(N)
width = np.pi/4*np.random.rand(N)
bars = ax.bar(theta, radii, width=width, bottom=0.0)
for r,bar in zip(radii, bars):
    bar.set_facecolor(cm.jet(r/10.))
    bar.set_alpha(0.5)

# Using dictionaries, create a color scheme for the text boxes
bbox_args = dict(boxstyle="round, pad=0.9", fc="green", alpha=0.5)
bbox_white = dict(boxstyle="round, pad=0.9", fc="1", alpha=0.9)
patch_white = dict(boxstyle="round, pad=1", fc="1", ec="1")

# Create various boxes with text annotations in them at specific
# x and y coordinates
ax.annotate(" ",
            xy=(.5,.93), ❺
```

```

xycoords='figure fraction', ❸
ha="center", va="center", ❷
bbox=patch_white) ❹

ax.annotate('Matplotlib and the Python Ecosystem for Scientific Computing',
            xy=(.5,.95),
            xycoords='figure fraction',
            xytext=(0, 0), textcoords='offset points',
            size=15,
            ha="center", va="center",
            bbox=bbox_args)

ax.annotate('Author and Lead Developer \n of Matplotlib ',
            xy=(.5,.82),
            xycoords='figure fraction',
            xytext=(0, 0), textcoords='offset points',
            ha="center", va="center",
            bbox=bbox_args)

ax.annotate('John D. Hunter',
            xy=(.5,.89),
            xycoords='figure fraction',
            xytext=(0, 0), textcoords='offset points',
            size=15,
            ha="center", va="center",
            bbox=bbox_white)

ax.annotate('Friday November 5th \n 2:00 pm \n1106ME ',
            xy=(.5,.25),
            xycoords='figure fraction',
            xytext=(0, 0), textcoords='offset points',
            size=15,
            ha="center", va="center",
            bbox=bbox_args)

ax.annotate('Sponsored by: \n The Hacker Within, \n'
            'The University Lectures Committee, \n The Department of '
            'Medical Physics\n and \n The American Nuclear Society',
            xy=(.78,.1),
            xycoords='figure fraction',
            xytext=(0, 0), textcoords='offset points',
            size=9,
            ha="center", va="center",
            bbox=bbox_args)

fig.savefig("plot.pdf")

```

- ❶ This common feature of executable Python scripts alerts the computer which Python to use.
- ❷ This imports the `colormaps` library from `matplotlib`.

- ③ This imports other libraries (`color`, `figure`, `show`, `rc`) from `matplotlib`.
- ④ This imports ellipse shapes from `matplotlib` (to be used as text boxes).
- ⑤ This creates an annotation box at certain x and y coordinates.
- ⑥ Those coordinates should be read as fractions of the figure height and width.
- ⑦ The horizontal and vertical text should be aligned to the center of the box.
- ⑧ The box being placed here should be white.

By executing the script (with `python scriptname.py`), everyone who received the email could produce the flyer shown in [Figure 7-5](#) using `matplotlib`, the topic of the seminar.

It was a very proud moment for this author when John said he liked the flyer in [Figure 7-5](#). After all, `matplotlib` was a key ingredient at that time not only in many dissertations, but also in the success of scientific Python. When John passed away in 2012, Fernando Perez described his contribution to the scientific computing community this way:

In 2002, John was a postdoc at the University of Chicago hospital working on the analysis of epilepsy seizure data in children. Frustrated with the state of the existing proprietary solutions for this class of problems, he started using Python for his work, back when the scientific Python ecosystem was much, much smaller than it is today and this could have been seen as a crazy risk. Furthermore, he found that there were many half-baked solutions for data visualization in Python at the time, but none that truly met his needs. Undeterred, he went on to create `matplotlib` and thus overcome one of the key obstacles for Python to become the best solution for open source scientific and technical computing.

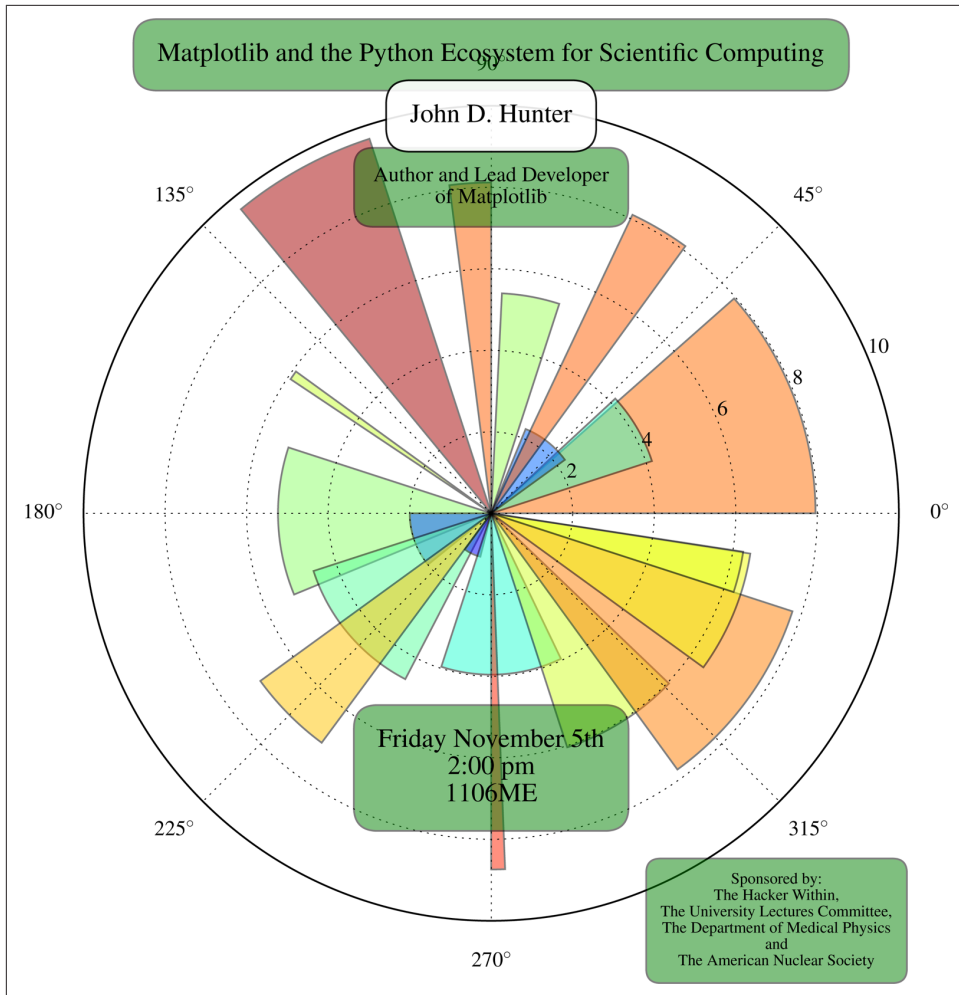


Figure 7-5. Announcement of 2010 John D. Hunter talk

Despite the loss of its creator, matplotlib continues to be improved by a vibrant team of extraordinary people, including Michael Droetboom, who now leads matplotlib development.

Additionally, matplotlib has provided a framework that other tools are capable of linking with. One such tool that is compatible with matplotlib is Bokeh.

Bokeh

Bokeh is a very simple, matplotlib-friendly API that can generate interactive plots appropriate for the Web. It abstracts somewhat from the matplotlib syntax, making

the user experience cleaner. The following is a script to plot the decay data as an HTML file using Bokeh:

decay_bokeh.py

```
import numpy as np
# import the Bokeh plotting tools
from bokeh import plotting as bp

# as in the matplotlib example, load decays.csv into a NumPy array
decaydata = np.loadtxt('decays.csv', delimiter=",", skiprows=1)

# provide handles for the x and y columns
time = decaydata[:,0]
decays = decaydata[:,1]

# define some output file metadata
bp.output_file("decays.html", title="Experiment 1 Radioactivity")

# create a figure with fun Internet-friendly features (optional)
bp.figure(tools="pan,wheel_zoom,box_zoom,reset,panzoomsave")

# on that figure, create a line plot
bp.line(time, decays, x_axis_label="Time (s)", y_axis_label="Decays (#)",
        color='#1F78B4', legend='Decays per second')

# additional customization to the figure can be specified separately
bp.curplot().title = "Decays"
bp.grid().grid_line_alpha=0.3

# open a browser
bp.show()
```

While Bokeh can produce plots in many formats, it was intended to produce interactive plots for viewing in a browser. Thus, when this script is run with `python decay_bokeh.py`, a browser opens automatically to display the interactive, pannable, zoomable plot in [Figure 7-6](#).

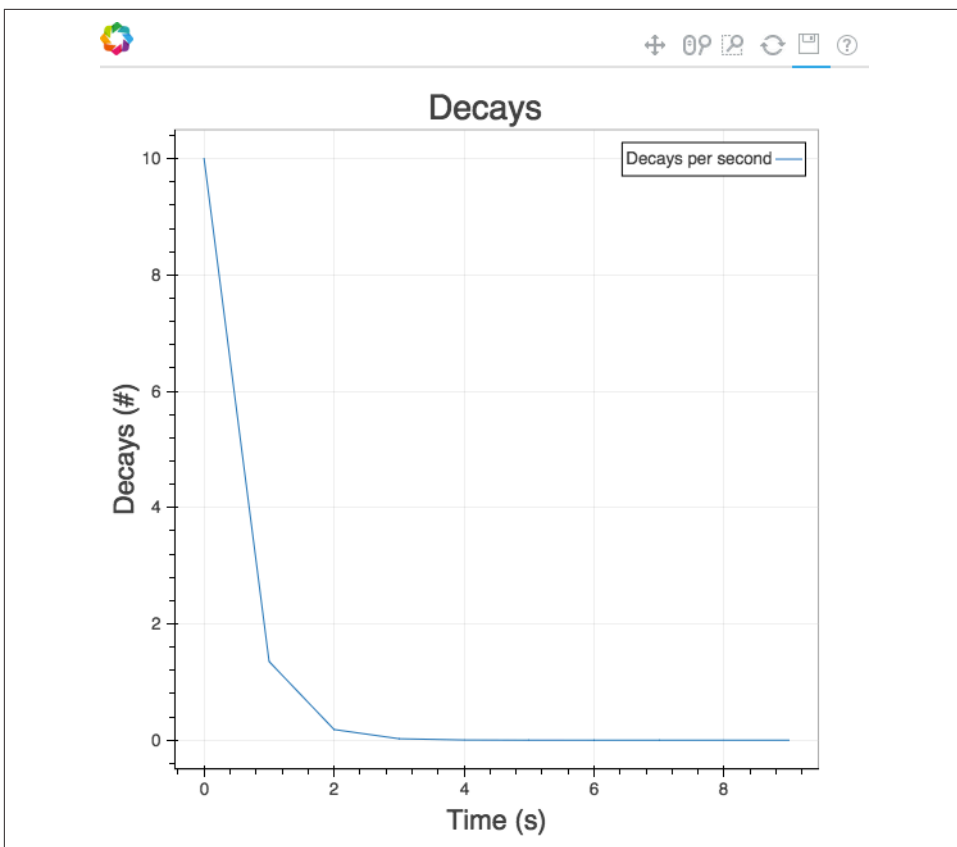


Figure 7-6. Decay plot with Bokeh

Bokeh is an excellent, easy-to-use tool for plots intended for publication on the Web. However, if the plot is complex or is intended for print media, matplotlib or gnuplot will serve that need better.

A key benefit of Bokeh, matplotlib, and gnuplot is their scriptable reproducibility. These tools are therefore the right choice for creating plots that do not yet exist. However, sometimes it is necessary to edit, crop, annotate, or otherwise manipulate an image file that already exists. The scripting tools in this section are all fully capable of handling these tasks. For cases when the original plot or diagram was not made with one of these tools, Inkscape is a good option for the physicist who needs to quickly tweak an image.

Inkscape

Inkscape is an open source project for producing and editing scalable vector graphics. Vector graphics are incredibly powerful. Rather than being pixelated, in a fixed-

resolution manner, scalable vector graphics are just that—scalable. Inkscape can be difficult to automate, however, because it is a tool that relies on a graphical user interface for manipulating plots by hand. Of course, this should be done only as a last resort because doing anything by hand does not scale, even if your resulting graphics do.

Analysis and Visualization Wrap-up

The vastness of the fleet of data analysis and visualization tools available today is staggering, and new tools appear at a blinding rate. By walking through a common workflow ([Figure 7-1](#)) that plotted the course from raw data to visualization, this chapter briefly introduced some of the tools available for:

- Loading data
- Cleaning data
- Wrangling data
- Analyzing data
- Plotting data

However, we have only covered the bare minimum of available analysis and visualization tools. The intention here was simply to provide you with a notion of the landscape of tasks at hand and tools available to perform them. The following chapters will go into much more depth concerning NumPy ([Chapter 9](#)), storing data ([Chapter 10](#)), and data structures ([Chapter 11](#)).

This chapter covered a few general guidelines that can be followed concerning data loading and preparation:

- Plain-text numerical data of a reasonable size should be loaded with NumPy.
- HDF5 data should be loaded with PyTables or Pandas.
- Pandas is useful for most everything else, especially munging.
- Data can be converted between formats using Blaze.

Visualization is key to presenting the results of your analyses, and before making a decision about what visualization tools to use we recommend that you closely observe the publication-quality plots in your particular subfield of the physical sciences, to get an idea of what tools are typically used and what features are included. More complex data visualizations were out of scope for this chapter. However, many tools for complex visualizations exist. In particular, for physicists with volumetric or higher-dimensional datasets, we recommend sophisticated Python tools for volumetric data such as `yt` and `mayavi`.

While the ideal tool varies depending on characteristics of the data, workflow, and goals, keep in mind that some things are universal:

- Beautiful is better than ugly.
- Simple is better than complex.
- Complex is better than complicated.
- Clear is better than dense.
- Communicating is better than convincing.
- Text must not distract.
- People are not mantis shrimp.

Publication

In science one tries to tell people, in such a way as to be understood by everyone, something that no one ever knew before. But in poetry, it's the exact opposite.

—Paul Dirac

One day, I'll find the right words, and they will be simple.

—Jack Kerouac

Publishing is an integral part of science. Indeed, the quality, frequency, and impact of publication records make or break a career in the physical sciences. Publication can take up an enormous fraction of time in a scientific career. However, with the right tools and workflow, a scientist can reduce the effort spent on mundane details (e.g., formatting, reference management, merging changes from coauthors) in order to spend more time on the important parts (e.g., literature review, data analysis, writing quality). This chapter will emphasize tools that allow the scientist to more efficiently accomplish and automate the former in order to focus on the latter. It will cover:

- An overview of document processing paradigms
- Employing text editors in document editing
- Markup languages for text-based document processing
- Managing references and automating bibliography creation

The first of these topics will be an overview of two competing paradigms in document processing.

Document Processing

Once upon a time, *Homo sapiens* etched our thoughts into stone tablets, on papyrus, and in graphical document processing software. All of these early tools share a com-

monality. They present the author with a “What You See Is What You Get” (WYSIWYG) paradigm in which formatting and content are inextricably combined. While this paradigm is ideal for artistic texts and documents with simple content, it can distract from the content in a scientific context and can make formatting changes a non-trivial task. Additionally, the binary format of most WYSIWYG documents increases the difficulty of version control, merging changes, and collaborating on a document.

Common document processing programs include:

- Microsoft Word
- Google Docs
- Open Office
- Libre Office

Though these tools have earned an important seat at the table in a business environment, they lack two key features that assist in efficient, reproducible paper-writing workflows. The first shortcoming is that they fail to separate the content (text and pictures) from the formatting (fonts, margins, etc.).

Separation of Content from Formatting

In the context of producing a journal publication, formatting issues are purely a distraction. In a WYSIWYG word processor, the act of choosing a title is polluted by the need to consider font, placement, and spacing. In this way, WYSIWYG editors fail to separate content from formatting. They thus prevent the author from focusing on word choice, clarity of explanation, and logical flow.

Furthermore, since each journal requires submissions to follow unique formatting guidelines, an efficient author avoids formatting concerns until a journal home is chosen. The wise author separates the content from the formatting since that choice may even be reevaluated after a rejection.

For all of these reasons, this chapter recommends a What You See Is What You Mean (WYSIWYM) document processing system for scientific work. Most such systems are plain text-based and rely on markup languages. Some common systems include:

- [LaTeX](#)
- [DocBook](#)
- [AsciiDoc](#)
- [PanDoc](#)

Among these, this chapter recommends the LaTeX context. Due to its powerful interface, beautiful mathematical typesetting, and overwhelming popularity in the physical

sciences, LaTeX is a fundamental skill for the effective researcher in the physical sciences.

Plain-text WYSIWYM tools such as these cleanly separate formatting from content. In a LaTeX document, layout specifications and formatting choices can be placed in a completely separate plain-text file than the ones in which the actual content of the paper is written. Because of this clean separation, switching from a document layout required by journal A to the layout required by journal B is done by switching the style files accordingly. The content of the paper is unaffected.

Additionally, this clean separation can enable efficient reference management. In the LaTeX context, this chapter will cover how this is achieved with bibliography files.

One more reason we recommend WYSIWYM editors over WYSIWYG document-processing tools is related to reproducibility: they facilitate tracking changes.

Tracking Changes

At the advent of computing, all information was stored as plain text. Now, information is stored in many complex binary formats. These binary formats are difficult to version control, since the differences between files rarely make logical sense without binary decoding. Many WYSIWYG document processors rely on such binary formats and are therefore difficult to version control in a helpful way.

Of course, an adept user of Microsoft Word will know that changes in that program can be tracked using its internal proprietary track-changes tool. While this is a dramatic improvement and enables concurrent efforts, more transparent and robust versioning can be achieved with version-controlled plain-text files. Since Microsoft's model requires that one person maintain the master copy and conduct the merges manually, concurrent editing by multiple parties can become untenable.

At this point in the book, you should have an appreciation of the effectiveness, efficiency, and provenance of version-controllable plain-text formats. Of course, for the same reasons, *we strongly recommend the use of a plain-text markup language for document processing*. Accordingly, we also recommend choosing a text editor appropriate for editing plain-text markup.

Text Editors

The editor used to write code can also be used to write papers. As mentioned in **Chapter 1**, text editors abound. Additionally, most text editors are very powerful. Accordingly, it can be a challenge to become proficient in the many features of more than one text editor. When new programmers seek to make an informed decision about which text editor to learn and use, many well-meaning colleagues may try to influence their choice.

However, these well-meaning colleagues should be largely ignored. A review of the features of a few common text editors (e.g., *vi*, *emacs*, *eclipse*, *nano*) should sufficiently illuminate the features and drawbacks of each.



Another argument for the use of plain-text markup is exactly this array of available text editors. That is, the universality of plain-text formatting and the existence of an array of text editors allows each collaborator to choose a preferred text editor and still contribute to the paper. On the other hand, with WYSIWYG, proprietary formats require that everyone must use the same tool.

Your efficiency with your chosen editor is more important than which text editor you choose. Most have a basic subset of tools (or available plug-ins) to accomplish:

- Syntax highlighting
- Text expansion
- Multiple file buffers
- Side-by-side file editing
- In-editor shell execution

Technical writing in a text editor allows the distractions of formatting to be separated from the content. To achieve this, the elements of a document are simply “marked up” with the special text syntax of a markup language.

Markup Languages

Markup languages provide syntax to annotate plain-text documents with structural information. A *build* step then produces a final document by combining that textual content and structural information with separate files defining styles and formatting. Most markup languages can produce multiple types of document (i.e., letters, articles, presentations) in many output file formats (*.pdf*, *.html*).

The ubiquitous HyperText Markup Language (HTML) may provide a familiar example of this process. Plain-text HTML files define title, heading, paragraph, and link elements of a web page. Layouts, fonts, and colors are separately defined in CSS files. In this way, web designers can focus on style (CSS) while the website owner can focus on the content (HTML).

This chapter will focus on the LaTeX markup language because it is the standard for publication-quality documents in the physical sciences. However, it is not the only available markup language. A few notable markup languages include:

- LaTeX

- Markdown
- reStructuredText
- MathML
- OpenMath

Markdown and reStructuredText both provide a simple, clean, readable syntax and can generate output in many formats. Python and GitHub users will encounter both formats, as reStructuredText is the standard for Python documentation and Markdown is the default markup language on GitHub. Each has syntax for including and rendering snippets of LaTeX. MathML and its counterpart OpenMath are optional substitutes for LaTeX, but lack its powerful extensions and wide adoption.

In markup languages, the term *markup* refers to the syntax for denoting the structure of the content. Content structure, distinct from formatting, enriches plain-text content with meaning. Directives, syntactically defined by the markup language, denote or *mark up* titles, headings, and sections, for example. Similarly, special characters mark up document elements such as tables, links, and other special content. Finally, rather than working in a single huge document, most markup languages enable constructing a document from many subfiles. In this way, complex file types, like images, can remain separate from the textual content. To include an image, the author simply references the image file by providing its location in the filesystem. In this way, the figures in a paper can remain in their native place on the filesystem and in their original file format. They are only pulled into the final document during the build step.

The build step is governed by the processing tool. For HTML, the tool is your browser. For the LaTeX markup language, however, it is the LaTeX program. The next section will delve deeper into LaTeX.

LaTeX

LaTeX (pronounced lay-tekh or lah-tekh) is the standard markup language in the physical sciences. Based on the TeX literate programming language, LaTeX provides a markup syntax customized for the creation of beautiful technical documents.

At a high level, a LaTeX document is made up of distinct constituent parts. The main file is simply a text file with the *.tex* file extension. Other LaTeX-related files may include style files (*.sty*), class files (*.cls*), and bibliography files (*.bib*). However, only the *.tex* file is necessary. That file need only contain four lines in order to constitute a valid LaTeX document. The first line chooses the type of document to create. This is called the LaTeX *document class*.

LaTeX document class

The first required line defines the type of document that should result. Common default options include `article`, `book`, and `letter`. The syntax is:

```
\documentclass{article}
```

This is a typical LaTeX command. It has the format:

```
\commandname[options]{argument}
```

The `documentclass` type should be listed in the curly braces. Options concerning the paper format and the font can be specified in square brackets before the curly braces. However, they are not necessary if the default styles are desired.

Note that many journals provide something called a *class file* and sometimes a *style file*, which contain formatting commands that comply with their requirements. The class file fully defines a LaTeX document *class*. So, for example, the journal publisher Elsevier provides an `elsarticle` document class. In order to convert any article into an Elsevier-compatible format, simply download the *elsarticle.cls* file to the directory containing the *.tex* files, and change the `documentclass` command argument to `elsarticle`. The rest of the document can stay the same.

The next two necessary lines are the commands that begin and end the document *environment*.

LaTeX environments

LaTeX environments are elements of a document. They can contain one another, like Russian dolls, and are denoted with the syntax:

```
\begin{environment} ... \end{environment}
```

`\begin{environment}` and `\end{environment}` are the commands that indicate environments in LaTeX. The top-level environment is the document environment. The document class, packages used, new command definitions, and other metadata appear before the document environment begins. This section is called the *preamble*. Everything after the document environment ends is ignored. For this reason, the `\begin{document}` command and the `\end{document}` command must each appear exactly once:

```
\documentclass{article}
\begin{document}
\end{document}
```

Since all actual content of the document appears within the document environment, between the `\begin{document}` and `\end{document}` commands, the shortest possible valid LaTeX file will include just one more line, a line of content!

```
\documentclass{article}
\begin{document}
Hello World!
\end{document}
```

This is a completely valid LaTeX document. Note that no information about fonts, document layout, margins, page numbers, or any other formatting details need clutter this document for it to be valid. However, it is only plain text right now. To render this text as a PDF, we must build the document.

Building the document

If the preceding content is placed into a document—say, *hello.tex*—a PDF document can be generated with two commands. The first runs the *latex* program, which compiles and renders a *.dvi* file. The second converts the *.dvi* file to the portable document format *.pdf*:

```
$ latex hello.tex ❶
$ dvipdf hello.dvi ❷
```

- ❶ LaTeX uses the *.tex* file to create a *.dvi* file.
- ❷ The *.dvi* file can be directly converted to *.pdf* with *dvipdf*.

Alternatively, if *pdflatex* is installed on your computer, that command can be used to accomplish both steps at once:

```
$ pdflatex hello.tex
```

As shown in **Figure 20-1**, the document is complete and contains only the text “Hello World!”

Hello World!

Figure 20-1. Hello World!

Now that the simplest possible document has been created with LaTeX, this chapter can move on to using LaTeX to produce publication-quality scientific documents. The first step will be to show how to appropriately mark up metadata elements of the document, such as the author names and title.

LaTeX metadata

Document metadata, such as the title of the document and the name of the author, may appear in many places in the document, depending on the layout. To make these special metadata variables available to the whole document, we define them in a scope outside of the document environment. The preamble holds information that

can help to define the document; it typically includes, at the very minimum, a `\title{*}` and `\author{}`, but can include other information as well.

When Ada Lovelace, often cited as history's first computer programmer, began to write the first mathematical algorithm, she wrote it in impeccable Victorian handwriting on reams of paper before it was typeset and reprinted by a printing press. This algorithm appeared in the appendices of a detailed technical description of its intended computer, Charles Babbage's Analytical Engine. The document itself, cleverly crafted in the span of nine months, contained nearly all the common features of a modern article in the physical sciences. It was full of mathematical proofs, tables, logical symbols, and diagrams. Had she had LaTeX at her disposal at the time, Ada might have written the document in LaTeX. She would have begun the document with metadata in the preamble as seen here:

```
% notes.tex ❶
\documentclass[11pt]{article} ❷

\author{Ada Augusta, Countess of Lovelace} ❸
\title{Notes By the Translator Upon the Memoir: Sketch of the Analytical Engine
Invented by Charles Babbage} ❹
\date{October, 1842} ❺
\begin{document} ❻
\maketitle ❼
\end{document} ❽
```

- ❶ In LaTeX, comments are preceded by a percent symbol.
- ❷ Ada would like to create an article-type document in 11pt font.
- ❸ She provides her formal name as the author metadata.
- ❹ She provides the full title.
- ❺ Another piece of optional metadata is the date.
- ❻ The document environment begins.
- ❼ The `\maketitle` command is executed. It uses the metadata to make a title.
- ❽ The document environment ends.

Notes By the Translator Upon the Memoir: Sketch of the Analytical Engine Invented by Charles Babbage

Ada Augusta, Countess of Lovelace

October, 1842

Figure 20-2. A Title in LaTeX

Ada's name, as well as the title of the article, should be defined in the preamble. However, they are only rendered into a main heading in the document with the use of the `\maketitle` command, which takes no arguments and must be executed within the document environment. The document that is produced appears in [Figure 20-2](#).



Exercise: Create a Document with Metadata

1. Create the *notes.tex* file in the previous code listing.
2. Run `latex notes.tex` and `dvipdf notes.tex` to create a *.pdf*.
3. View it.
4. Remove the value for the date so that it reads `\date{}`.
5. Repeat steps 2 and 3. What changed?

Now that the basics are clear, scientific information can be added to this document. In support of that, the document will need some underlying structure, such as sections and subsections. The next section will show how LaTeX markup can be used to demarcate those structural elements of the document.

Document structure

In the body of the document, the document structure is denoted by commands declaring the titles of each structural element. In the `article` document class, these include sections, subsections, subsubsections, and paragraphs. In a book, the structure includes parts and chapters as well. Ada's foundational notes were lettered A through G. The body of her document, therefore, would have included one `\section` command for each section:

```
% notes.tex
\documentclass[11pt]{article}
```

```

\author{Ada Augusta, Countess of Lovelace}
\title{Notes By the Translator Upon the Memoir: Sketch of the Analytical Engine
Invented by Charles Babbage}
\date{October, 1842}
\begin{document}
\maketitle

\section{Note A}
\section{Note B}
\section{Note C}
\section{Note D}
\section{Note E}
\section{Note F}
\section{Note G}

\end{document}

```

Since each note is a separate entity, however, it may be wise for Ada to keep them in separate files to simplify editing. In LaTeX, rather than keeping all the sections in one big file, Ada can include other LaTeX files in the master file. If the content of Note A, for example, is held in its own *intro.tex* file, then Ada can include it with the `\input{}` command. In this way, sections can be moved around during the editing process with ease. Additionally, the content is then stored in files named according to meaning rather than document order:

```

\section{Note A}
\input{intro}

\section{Note B}
\input{storehouse}

...

\section{Note G}
\input{conclusion}

```

Any text and LaTeX syntax in *intro.tex* will be inserted by LaTeX at the line where the command appeared. This multiple-file-inclusion paradigm is very powerful and encourages the reuse of document subparts. For example, the text that acknowledges your grant funding can stay in just one file and can be simply `\input` into each paper.

Now that the document has a structure, we can get to work filling in the text and equations that make up the content of the paper. That will utilize the most important capability in LaTeX: typesetting math.

Typesetting mathematical formulae

LaTeX's support for mathematical typesetting is unquestionably the most important among its features. LaTeX syntax for typesetting mathematical formulae has set the standard for technical documents. Publication-quality mathematical formulae must

include beautifully rendered Greek and Latin letters as well as an enormous array of logical, mathematical symbols. Beyond the typical symbols, LaTeX possesses an enormous library of esoteric ones.

Some equations must be rendered inline with paragraph text, while others should be displayed on a separate line. Furthermore, some must be aligned with one another, possess an identifying equation number, or incorporate interleaved text, among other requirements. LaTeX handles all of these situations and more.

To render math symbols or equations inline with a sentence, LaTeX math mode can be denoted with a simple pair of dollar signs (\$). Thus, the LaTeX syntax shown here is rendered as in **Figure 20-3**:

The particular function whose integral the Difference Engine was constructed to tabulate, is $\Delta^7 u_x = 0$. The purpose which that engine has been specially intended and adapted to fulfil, is the computation of nautical and astronomical tables. The integral of $\Delta^7 u_x = 0$ being $u_z = a + bx + cx^2 + dx^3 + ex^4 + fx^5 + gx^6$, the constants a, b, c, &c. are represented on the seven columns of discs, of which the engine consists.

Note the dollar signs denoting the beginning and end of each inline mathematical equation. In an equation, mathematical markup can be used. Symbols, like the capital Greek letter delta, are denoted with a backslash. The caret (^) indicates a following superscript, and an underscore (_) means subscript.

The particular function whose integral the Difference Engine was constructed to tabulate, is $\Delta^7 u_x = 0$. The purpose which that engine has been specially intended and adapted to fulfil, is the computation of nautical and astronomical tables. The integral of $\Delta^7 u_x = 0$ being $u_z = a + bx + cx^2 + dx^3 + ex^4 + fx^5 + gx^6$, the constants a, b, c, etc. are represented on the seven columns of discs, of which the engine consists.

Figure 20-3. Inline equations

Alternatively, to display one or more equations on a line separated from the text, an equation-focused LaTeX environment is used:

In fact the engine may be described as being the material expression of any indefinite function of any degree of generality and complexity, such as for instance,

```
\begin{equation} ❶  
F(x, y, z, \log x, \sin y, x^p), ❷  
\end{equation}
```

which is, it will be observed, a function of all other possible functions of any number of quantities.

- ❶ An equation environment denotes an equation separated from the text, nicely centered.
- ❷ In this environment, mathematical markup can be used.

The equation is thereby drawn out of the text and is automatically given an equation number, as in [Figure 20-4](#).

The following is a more complicated example of the manner in which the engine would compute a trigonometrical function containing variables. To multiply

$$F(x, y, z, \log x, \sin y, x^p) \quad (1)$$

which is, it will be observed, a function of all other functions of any number of quantities.

Figure 20-4. The equation environment

LaTeX enables a multitude of such mathematical typesetting conventions common to publications in the physical sciences. For example, multiple equations can be beautifully aligned with one another using the `align` math environment and ampersands (&) to mark the point of alignment. The American Mathematical Society made this possible by creating a package that it has made available to LaTeX users. To use this aligning environment, Ada will have to load the appropriate package when running LaTeX. That is done in the preamble.

Packages Extend LaTeX Capabilities

In addition to metadata, the preamble often declares the inclusion of any packages that the document relies on. Standard packages include `amsmath`, `amsfonts`, and `amssymb`. These are the American Mathematical Society packages for math layouts, math fonts, and math symbols, respectively. Another common package is `graphicx`, which allows the inclusion of *.eps* figures.

The `align` environment is available in the `amsmath` package, so if Ada wants to use it, she must include that package in her preamble. To enable an extended library of symbols, she might also include the `amssymb` package. Finally, since the description of the Bernoulli algorithm for the Analytical Engine required enormous, detailed tables spanning many pages, Ada might have also wanted to use the `longtable` package, which enables flexible tables that break across pages. Here are the lines she'll need to add to her preamble:

```
\usepackage{amsmath}
\usepackage{amssymb}
\usepackage{longtable}
```

If she has added the `amsmath` package to her preamble, Ada can thus make beautifully aligned equations with the `align` environment, as in the snippet shown here (rendered in [Figure 20-5](#)):

The following is a more complicated example of the manner in which the engine would compute a trigonometrical function containing variables.
To multiply

```
\begin{align}
&A+A_1 \cos \theta + A_2 \cos^2 \theta + A_3 \cos^3 \theta + \dots \\
&\intertext{by} \\
&B+B_1 \cos \theta . \\
\end{align}
```

- 1 The ampersand marks the place in this equation that should line up with the next equation.
- 2 The progression of mathematical operations can be documented with common interleaved phrases such as “where,” “such that,” or “which reduces to.” To interleave such text in a math environment, the `\intertext` command is used.
- 3 The ampersand in the second equation marks the place in this equation that lines up with the ampersand of the first equation.

The following is a more complicated example of the manner in which the engine would compute a trigonometrical function containing variables. To multiply

$$A + A_1 \cos \theta + A_2 \cos 2\theta + A_3 \cos 3\theta + \quad (1)$$

by

$$B + B_1 \cos \theta. \quad (2)$$

Figure 20-5. *Aligned LaTeX*

As you can see, the equations line up just where the ampersands were placed, but the ampersands do not appear in the rendered document. Of course, this is only a taste of mathematical typesetting capabilities in LaTeX, and equations are only half the battle.

How does LaTeX handle other elements of technical documents, such as tables and figures?

Tables and figures

Tables and figures also often belong in their own files. In addition to the simplicity gained by keeping such elements outside of the text, reusing these elements in other documents becomes much simpler if they are kept in their own files. LaTeX is capable (with the `beamer` package) of generating presentation-style documents, and these files can be reused in those documents with a simple reference.

By keeping the figures themselves out of the main text file, the author can focus on the elements of the figures that are related to the flow of the document: placement relative to the text, captions, relative size, etc.

In Ada's notes, diagrams of variables related to the algorithm were inserted. These could be created in a LaTeX math environment, but they could also be included as figures. The syntax for including an image is:

```
\begin{figure}[htbp] ❶
\begin{center} ❷
\includegraphics[width=0.5\textwidth]{var_diagram} ❸
\end{center}
\caption{Any set of columns on which numbers are inscribed, represents
merely a general function of the several quantities, until the special
function have been impressed by means of the Operation and
Variable-cards.} ❹
\label{fig:var_diagram} ❺
\end{figure}
```

- ❶ The figure environment begins. Placement options are specified as (h)ere, (t)op, (b)ottom, or on its own (p)age.
- ❷ The figure should appear horizontally centered on the page.
- ❸ The name of the image file is indicated and the `width` option specifies that the figure should be half the width of the text.
- ❹ A verbose caption is added.
- ❺ A label is added so that the figure can be referenced later in the document using this name tag.

The result of this syntax is shown in **Figure 20-6**. In it, the image is brought into the document, numbered, sized, and captioned exactly as was meant.

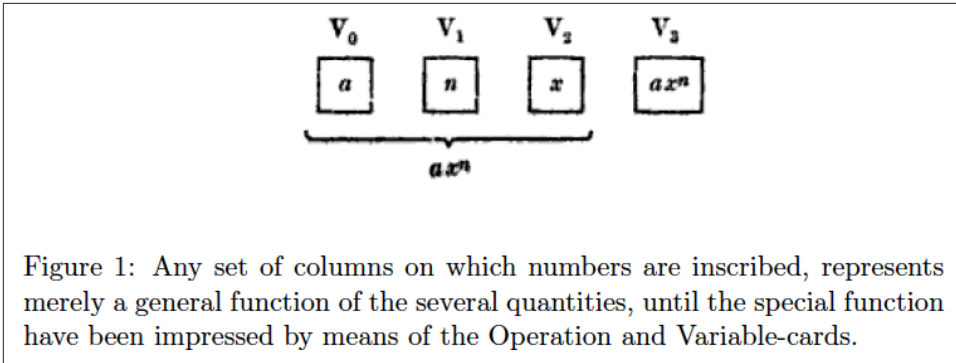


Figure 20-6. Labels in LaTeX

In this example, the figure was labeled with the `\label` command so that it can be referenced later in the document. This clean, customizable syntax for internal references is a feature of LaTeX that improves efficiency greatly. The next section will show how such internal references work.

Internal references

The LaTeX syntax for referencing document elements such as equations, tables, figures, and sections entirely eliminates the overhead of matching equation and section numbers with their in-text references. The `\ref{}` command can be embedded in the text to refer to these elements elsewhere in the document if the elements have been labeled with the corresponding `\label{}` command.

At build time, LaTeX numbers the document elements and expands inline references accordingly. Since tables, figures, and sections are often reordered during the editing process, referring to them by meaningful labels is much more efficient than trying to keep track of meaningless numbers. In Note D (*example.tex*), for instance, Ada presents a complex example and refers to Note B (*storehouse.tex*). Since the ordering of the notes might not have been finalized at the time of writing, referring to Note B by a meaningful name rather than a number—or, in this case, a letter—is preferred. To do this, Ada must use the `\label{}` command in the *storehouse.tex* file so that the *example.tex* file may refer to it with the `\ref{}` command:

```
% storehouse.tex ❶
\label{sec:storehouse}❷
That portion of the Analytical Engine here alluded to is called the
storehouse. . .
```

- ❶ The section on the storehouse is stored in a file called *storehouse.tex*.
- ❷ That section is called Note B, but Ada remembers it as the storehouse section, and labels it such.

In this example, the label uses the prefix `sec:` to indicate that *storehouse* is a section. This is not necessary. However, it is a common and useful convention. Similarly, figures are prepended with `fig:`, tables with `tab:`, and so on. Ada can now reference Note B from Note D as shown here:

```
% example.tex ❶
We have represented the solution of these two equations below, with
every detail, in a diagram similar to those used in Note
\ref{sec:storehouse}; ... ❷
```

❶ Note D is held in *example.tex*.

❷ Ada can reference Note B within this file using the memorable label.

When the document is built with these two files, a “B” will appear automatically where the reference is. The same can be achieved with figure and table labels as well. Now it is clear how to reference figures and sections. However, there is another kind of reference common in publications. Bibliographic references (citations) are also automated, but are handled a bit differently in LaTeX. The next section will explain how.

Bibliographies

An even more powerful referencing feature of LaTeX is its syntax for citation of bibliographic references and its automated formatting of bibliographies. Using BibTeX or BibLaTeX, bibliography management in LaTeX begins with *.bib* files. These contain information about resources cited in the text sufficient to construct a bibliography. Had Ada desired to cite the Scientific Memoir her notes were concerned with, she might have defined that work in a *refs.bib* file as follows:

```
% refs.bib
@article{menabrea_sketch_1842,
  series = {Scientific Memoirs},
  title = {Sketch of The Analytical Engine Invented by Charles Babbage},
  volume = {3},
  journal = {Taylor's Scientific Memoirs},
  author = {Menabrea, L.F.},
  month = oct,
  year = {1842},
  pages = {666--731}
}
```

To cite this work in the body of her text and generate an associated bibliography, Ada must do only three things. First, she uses the `\cite{}` command, along with the key (`menabrea_sketch_1842`), where she wants the reference to appear:

```
% intro.tex
...
These cards contain within themselves (in a manner explained in the Memoir
```

```
itself \cite{menabrea_sketch_1842}) the law of development of the particular
function that may be under consideration, and they compel the mechanism to act
accordingly in a certain corresponding order.
...
```

Second, she must include a command placing the bibliography. Bibliographies appear at the end of a document, so just before the `\end{document}` command in her main *notes.tex* file, Ada adds two lines:

```
% notes.tex
...

\section{Note G}
\input{conclusion}

\bibliographystyle{plain}
\bibliography{refs}
\end{document}
```

These together define the bibliography style. The choices for this parameter are myriad. The simplest choice is often “plain,” as has been used here. However, a one-word change can alter the formatting to comply with Chicago, MLA, or any other bibliography formatting style. The second line names the location(s) of the *.bib* file(s).

The final necessary step is to build the bibliography along with the document. For this, an extra build step is required that employs the `bibtex` command. In a peculiarity of LaTeX, for the references to appear, you must call `latex` again twice after issuing the `bibtex` command. So, at the command line, Ada must type:

```
$ latex notes
$ bibtex notes
$ latex notes
$ latex notes
$ dvipdf notes
```

The result is marvelous. In the text, the `\cite` command is replaced with “[1]”, and on the final page of her document, a bibliography appears as in [Figure 20-7](#).

References

- [1] L.F. Menabrea. Sketch of the analytical engine invented by charles babage. *Taylor’s Scientific Memoirs*, 3:666–731, October 1842.

Figure 20-7. Automated bibliography generation

Never again need scientists concern themselves with the punctuation after a title in an MLA-style bibliography—LaTeX has automated this. The only thing LaTeX does not automate about bibliography creation is reading the papers and making the *.bib*

file itself. Thankfully, other tools exist to make that process more efficient. The next section introduces these.

Reference management

To generate a *.bib* file easily, consider using a reference manager. Such a tool helps to collect and organize bibliographic references. By helping the researcher automate the collection of metadata about journal articles and other documents, as well as the production of *.bib* files, these tools eliminate the tedious task of typing names, titles, volume numbers, and dates for each reference cited in a paper. It can all be completely automated. A number of open source tools for this task exist. These include, among others:

- BibDesk
- EndNote
- JabRef
- Mendeley
- RefWorks
- Zotero

Reference managers help researchers to organize their sources by storing the metadata associated with them. That metadata can typically be exported as *.bib* files.

Citing Code and Data

One thing that BibTeX lacks is a metadata format appropriate for uniquely referencing code or data, unless it has a digital object identifier (DOI) number associated with it. For truly reproducible publication, you should cite the code and data that produced the analysis using a DOI.

Each commit in your version-controlled code repository has a commit hash number that distinguishes it uniquely from others. For unique identification in a library or bookstore, this book has an ISBN. Analogously, data and software objects can be identified in a persistent way with a DOI number.

It is possible to acquire a DOI for any piece of software using archival services on the Internet. Some are even free and open source.

The use of these reference managers is outside the scope of this chapter. Please go to the individual tools' websites to get started using them.

Publication Wrap-up

Publication is the currency of a scientific career. It is the traditional way in which scientific work is shared with and judged by our peers. For this reason, scientists spend a lot of time producing publication-quality documents. This chapter has sought to provide an overview of the tools available to aid you in this pursuit and to give an introduction to the most ubiquitous, LaTeX. Now that you have read this chapter, you should know that:

- Markup languages separate formatting from content.
- Markup-based text documents are more version-controllable.
- Many markup languages exist, but LaTeX is a particularly powerful tool for scientific publication.

In the context of LaTeX, you should also know how to:

- Produce a simple document
- Give structure to that document
- Add mathematical equations inline
- Display mathematics
- Include figures
- Reference those figures
- Cite bibliographic references
- Automate the creation of a bibliography

With these skills, you are equipped to begin generating lovely publication-quality documents. Many resources are available online to enrich what you have learned here. Two favorites are:

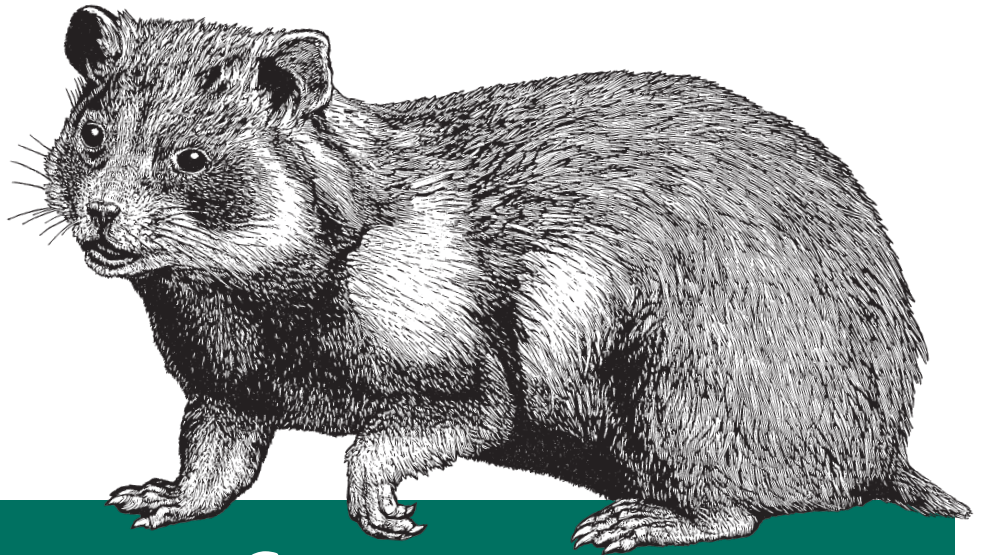
- [“The Not So Short Introduction to LaTeX”](#), by Tobias Oetiker et al.
- [Tex-LaTeX Stack Exchange](#)

IPython Notebook

As an aside, please note that another option for reproducible document creation that was not mentioned in this chapter (because it is in a class of its own) is the IPython notebook. IPython Notebook is a part of the IPython interpreter that has been used in previous chapters. It is an interface for Python that can incorporate markup languages and code into a reproducible document. With an interface very similar to that of a Mathematica notebook, the IPython (soon, Jupyter) notebook combines plain text, LaTeX, and other markup with code input and output cells. Since the IPython notebook displays tables and plots alongside the code that generated them, a document in this format is especially reproducible.

For more on IPython, Jupyter, and working with the Notebook, see [the IPython website](#).

Publication is an essential part of bringing your work to your peers. Another way, however, is direct collaboration. The next chapter will demonstrate how GitHub can make collaboration on papers and software far more efficient.



Bioinformatics Data Skills

REPRODUCIBLE AND ROBUST RESEARCH WITH OPEN SOURCE TOOLS

Vince Buffalo

Bioinformatics Data Skills

Vince Buffalo

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Working with Remote Machines

Most data-processing tasks in bioinformatics require more computing power than we have on our workstations, which means we must work with large servers or computing clusters. For some bioinformatics projects, it's likely you'll work predominantly over a network connection with remote machines. Unsurprisingly, working with remote machines can be quite frustrating for beginners and can hobble the productivity of experienced bioinformaticians. In this chapter, we'll learn how to make working with remote machines as effortless as possible so you can focus your time and efforts on the project itself.

Connecting to Remote Machines with SSH

There are many ways to connect to another machine over a network, but by far the most common is through the *secure shell* (SSH). We use SSH because it's encrypted (which makes it secure to send passwords, edit private files, etc.), and because it's on every Unix system. How your server, SSH, and your user account are configured is something you or your system administrator determines; this chapter won't cover these system administration topics. The material covered in this section should help you answer common SSH questions a sysadmin may ask (e.g., "Do you have an SSH public key?"). You'll also learn all of the basics you'll need as a bioinformatician to SSH into remote machines.

To initialize an SSH connection to a host (in this case, *biocluster.myuniversity.edu*), we use the `ssh` command:

```
$ ssh biocluster.myuniversity.edu
Password: ❶
Last login: Sun Aug 11 11:57:59 2013 from fisher.myisp.com
wsobchak@biocluster.myuniversity.edu$ ❷
```

- ❶ When connecting to a remote host with SSH, you'll be prompted for your remote user account's password.
- ❷ After logging in with your password, you're granted a shell prompt on the remote host. This allows you to execute commands on the remote host just as you'd execute them locally.

SSH also works with IP addresses—for example, you could connect to a machine with `ssh 192.169.237.42`. If your server uses a different port than the default (port 22), or your username on the remote machine is different from your local username, you'll need to specify these details when connecting:

```
$ ssh -p 50453 cdarwin@biocluster.myuniversity.edu
```

Here, we've specified the port with the flag `-p` and the username by using the syntax `user@domain`. If you're unable to connect to a host, using `ssh -v` (`-v` for verbose) can help you spot the issue. You can increase the verbosity by using `-vv` or `-vvv`; see `man ssh` for more details.



Storing Your Frequent SSH Hosts

Bioinformaticians are constantly having to SSH to servers, and typing out IP addresses or long domain names can become quite tedious. It's also burdensome to remember and type out additional details like the remote server's port or your remote username. The developers behind SSH created a clever alternative: the SSH config file. SSH config files store details about hosts you frequently connect to. This file is easy to create, and hosts stored in this file work not only with `ssh`, but also with two programs we'll learn about in [Chapter 6](#): `scp` and `rsync`.

To create a file, just create and edit the file at `~/.ssh/config`. Each entry takes the following form:

```
Host bio_serv
  HostName 192.168.237.42
  User cdarwin
  Port 50453
```

You won't need to specify `Port` and `User` unless these differ from the remote host's defaults. With this file saved, you can SSH into `192.168.236.42` using the alias `ssh bio_serv` rather than typing out `ssh -p 50453 cdarwin@192.169.237.42`.

If you're working with many remote machine connections in many terminal tabs, it's sometimes useful to be make sure you're working on the host you think you are. You can always access the hostname with the command `hostname`:

```
$ hostname
biocluster.myuniversity.edu
```

Similarly, if you maintain multiple accounts on a server (e.g., a user account for analysis and a more powerful administration account for sysadmin tasks), it can be useful to check which account you're using. The command `whoami` returns your username:

```
$ whoami
cdarwin
```

This is especially useful if you do occasionally log in with an administrator account with more privileges—the potential risks associated with making a mistake on an account with administrator privileges are much higher, so you should always be away when you're on this account (and minimize this time as much as possible).

Quick Authentication with SSH Keys

SSH requires that you type your password for the account on the remote machine. However, entering a password each time you login can get tedious, and not always safe (e.g., keyboard input could be monitored). A safer, easier alternative is to use an *SSH public key*. Public key cryptography is a fascinating technology, but the details are outside the scope of this book. To use SSH keys to log in into remote machines without passwords, we first need to generate a public/private key pair. We do this with the command `ssh-keygen`. It's very important that you note the difference between your public and private keys: you can distribute your public key to other servers, but your private key must be kept safe and secure and never shared.

Let's generate an SSH key pair using `ssh-keygen`:

```
$ ssh-keygen -b 2048
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/username/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/username/.ssh/id_rsa.
Your public key has been saved in /Users/username/.ssh/id_rsa.pub.
The key fingerprint is:
e1:1e:3d:01:e1:a3:ed:2b:6b:fe:c1:8e:73:7f:1f:f0
The key's randomart image is:
+--[ RSA 2048 ]-----+
|.O... ..|
| . . o |
| . * |
| . o + |
| . S . |
| o . E |
| + . |
|oo+.. . |
|+=oo... o.|
+-----+
```

This creates a private key at `~/.ssh/id_rsa` and a public key at `~/.ssh/id_rsa.pub`. `ssh-keygen` gives you the option to use an empty password, but it's generally recommended that you use a real password. If you're wondering, the random art `ssh-keygen` creates is a way of validating your keys (there are more details about this in `man ssh` if you're curious).

To use password-less authentication using SSH keys, first SSH to your remote host and log in with your password. Change directories to `~/.ssh`, and append the contents of your public key file (`id_rsa.pub`, *not* your private key!) to `~/.ssh/authorized_keys` (note that the `~` may be expanded to `/home/username` or `/Users/username` depending on the remote operating system). You can append this file by copying your public key from your local system, and pasting it to the `~/.ssh/authorized_keys` file on the remote system. Some systems have an `ssh-copy-id` command that automatically does this for you.

Again, be sure you're using your public key, and *not* the private key. If your private key ever is accidentally distributed, this compromises the security of the machines you've set up key-based authentication on. The `~/.ssh/id_rsa` private key has read/write permissions only for the creator, and these restrictive permissions should be kept this way.

After you've added your public key to the remote host, try logging in a few times. You'll notice that you keep getting prompted for your SSH key's password. If you're scratching your head wondering how this saves time, there's one more trick to know: `ssh-agent`. The `ssh-agent` program runs in the background on your local machine, and manages your SSH key(s). `ssh-agent` allows you to use your keys without entering their passwords each time—exactly what we want when we frequently connect to servers. SSH agent is usually already running on Unix-based systems, but if not, you can use `eval ssh-agent` to start it. Then, to tell `ssh-agent` about our key, we use `ssh-add`:

```
$ ssh-add
Enter passphrase for /Users/username/.ssh/id_rsa:
Identity added: /Users/username/.ssh/id_rsa
```

Now, the background `ssh-agent` process manages our key for us, and we won't have to enter our password each time we connect to a remote machine. I once calculated that I connect to different machines about 16 times a day, and it takes me about two seconds to enter my password on average (accounting for typing mistakes). If we were to assume I didn't work on weekends, this works out to about 8,320 seconds, or 2.3 hours a year of just SSH connections. After 10 years, this translates to nearly an entire day wasted on just connecting to machines. Learning these tricks may take an hour or so, but over the course of a career, this really saves time.

Maintaining Long-Running Jobs with nohup and tmux

In [Chapter 3](#), we briefly discussed how processes (whether running in the foreground or background) will be terminated when we close our terminal window. Processes are also terminated if we disconnect from our servers or if our network connection temporarily drops out. This behavior is intentional—your program will receive the *hangup* signal (referred to more technically as SIGHUP), which will in almost all cases cause your application to exit immediately. Because we’re perpetually working with remote machines in our daily bioinformatics work, we need a way to prevent hangups from stopping long-running applications. Leaving your local terminal’s connection to a remote machine open while a program runs is a fragile solution—even the most reliable networks can have short outage blips. We’ll look at two preferable solutions: nohup and Tmux. If you use a cluster, there are better ways to deal with hangups (e.g., submitting batch jobs to your cluster’s software), but these depend on your specific cluster configuration. In this case, consult your system administrator.

nohup

nohup is simple command that executes a command and catches hangup signals sent from the terminal. Because the nohup command is catching and ignoring these hangup signals, the program you’re running won’t be interrupted. Running a command with nohup is as easy as adding nohup before your command:

```
$ nohup program1 > output.txt & ❶  
[1] 10900 ❷
```

- ❶ We run the command with all options and arguments as we would normally, but by adding nohup this program will not be interrupted if your terminal were to close or the remote connection were to drop. Additionally, it’s a good idea to redirect standard output and standard error just as we did in [Chapter 3](#) so you can check output later.
- ❷ nohup returns the process ID number (or PID), which is how you can monitor or terminate this process if you need to (covered in “[Killing Processes](#)” on page 51). Because we lose access to this process when we run it through nohup, our only way of terminating it is by referring to it by its process ID.

Working with Remote Machines Through Tmux

An alternative to nohup is to use a *terminal multiplexer*. In addition to solving the hangup problem, using a terminal multiplexer will greatly increase your productivity when working over a remote connection. We’ll use a terminal multiplexer called

Tmux, but a popular alternative is GNU Screen. Tmux and Screen have similar functionality, but Tmux is more actively developed and has some additional nice features.

Tmux (and terminal multiplexers in general) allow you to create a session containing multiple windows, each capable of running their own processes. Tmux's sessions are persistent, meaning that all windows and their processes can easily be restored by reattaching the session.

When run on a remote machine, Tmux allows you to maintain a persistent session that won't be lost if your connection drops or you close your terminal window to go home (or even quit your terminal program). Rather, all of Tmux's sessions can be reattached to whatever terminal you're currently on—just SSH back into the remote host and reattach the Tmux session. All windows will be undisturbed and all processes still running.

Installing and Configuring Tmux

Tmux is available through most package/port managers. On OS X, Tmux can be installed through Homebrew and on Ubuntu it's available through `apt-get`. After installing Tmux, I strongly suggest you go to this chapter's directory on GitHub and download the `.tmux.conf` file to your home directory. Just as your shell loads configurations from `~/.profile` or `~/.bashrc`, Tmux will load its configurations from `~/.tmux.conf`. The minimal settings in `.tmux.conf` make it easier to learn Tmux by giving you a useful display bar at the bottom and changing some of Tmux's key bindings to those that are more common among Tmux users.

Creating, Detaching, and Attaching Tmux Sessions

Tmux allows you to have multiple *sessions*, and within each session have multiple windows. Each Tmux session is a separate environment. Normally, I use a session for each different project I'm working on; for example, I might have a session for maize SNP calling, one for developing a new tool, and another for writing R code to analyze some *Drosophila* gene expression data. Within each of these sessions, I'd have multiple windows. For example, in my maize SNP calling project, I might have three windows open: one for interacting with the shell, one with a project notebook open in my text editor, and another with a Unix manual page open. Note that all of these windows are *within* Tmux; your terminal program's concept of tabs and windows is entirely different from Tmux's. Unlike Tmux, your terminal cannot maintain persistent sessions.

Let's create a new Tmux session. To make our examples a bit easier, we're going to do this on our local machine. However, to manage sessions on a remote host, we'd need to start Tmux on that remote host (this is often confusing for beginners). Running Tmux on a remote host is no different; we just SSH in to our host and start Tmux

there. Suppose we wanted to create a Tmux session corresponding to our earlier Maize SNP calling example:

```
$ tmux new-session -s maize-snps
```

Tmux uses subcommands; the `new-session` subcommand just shown creates new sessions. The `-s` option simply gives this session a name so it's easier to identify later. If you're following along and you've correctly placed the `.tmux.conf` file in your home directory, your Tmux session should look like [Figure 4-1](#).



```
bash-3.2$ echo "hello, tmux"
hello, tmux
bash-3.2$
```

maize-snps 0:bash* "stebbins" 21:54 10-Feb-14

Figure 4-1. Tmux using the provided `.tmux.conf` file

Tmux looks just like a normal shell prompt except for the status bar it has added at the bottom of the screen (we'll talk more about this in a bit). When Tmux is open, we interact with Tmux through keyboard shortcuts. These shortcuts are all based on first pressing Control and *a*, and then adding a specific key after (releasing Control-*a* first). By default, Tmux uses Control-*b* rather than Control-*a*, but this is a change we've made in our `.tmux.conf` to follow the configuration preferred by most Tmux users.

The most useful feature of Tmux (and terminal multiplexers in general) is the ability to detach and reattach sessions without losing our work. Let's see how this works in Tmux. Let's first enter something in our blank shell so we can recognize this session later: `echo "hello, tmux"`. To detach a session, we use Control-*a*, followed by *d* (for detach). After entering this, you should see Tmux close and be returned to your regular shell prompt.

After detaching, we can see that Tmux has kept our session alive by calling `tmux` with the `list-sessions` subcommand:

```
$ tmux list-sessions
maize-snps: 1 windows (created Mon Feb 10 00:06:00 2014) [180x41]
```

Now, let's reattach our session. We reattach sessions with the `attach-session` subcommand, but the shorter `attach` also works:

```
$ tmux attach
```

Note that because we only have one session running (our `maize-snps` session) we don't have to specify which session to attach. Had there been more than one session running, all session names would have been listed when we executed `list-sessions` and we could reattach a particular session using `-t <session-name>`. With only one Tmux session running, `tmux attach` is equivalent to `tmux attach-session -t maize-snps`.

Managing remote sessions with Tmux is no different than managing sessions locally as we did earlier. The only difference is that we create our sessions on the remote host after we connect with SSH. Closing our SSH connection (either intentionally or unintentionally due to a network drop) will cause Tmux to detach any active sessions.

Working with Tmux Windows

Each Tmux session can also contain multiple windows. This is especially handy when working on remote machines. With Tmux's windows, a single SSH connection to a remote host can support multiple activities in different windows. Tmux also allows you to create multiple *panes* within a window that allow you to split your windows into parts, but to save space I'll let the reader learn this functionality on their own. Consult the Tmux manual page (e.g., with `man tmux`) or read one of the many excellent Tmux tutorials on the Web.

Like other Tmux key sequences, we create and switch windows using `Control-a` and then another key. To create a window, we use `Control-a c`, and we use `Control-a n` and `Control-a p` to go to the next and previous windows, respectively. [Table 4-1](#) lists the most commonly used Tmux key sequences. See `man tmux` for a complete list, or press `Control-a ?` from within a Tmux session.

Table 4-1. Common Tmux key sequences

Key sequence	Action
<code>Control-a d</code>	Detach
<code>Control-a c</code>	Create new window

Key sequence	Action
Control- <i>a n</i>	Go to next window
Control- <i>a p</i>	Go to previous window
Control- <i>a &</i>	Kill current window (exit in shell also works)
Control- <i>a ,</i>	Rename current window
Control- <i>a ?</i>	List all key sequences

Table 4-2 lists the most commonly used Tmux subcommands.

Table 4-2. Common Tmux subcommands

Subcommand	Action
<code>tmux list-sessions</code>	List all sessions.
<code>tmux new-session -s session-name</code>	Create a new session named "session-name".
<code>tmux attach-session -t session-name</code>	Attach a session named "session-name".
<code>tmux attach-session -d -t session-name</code>	Attach a session named "session-name", detaching it first.

If you use Emacs as your text editor, you'll quickly notice that the key binding Control-*a* may get in the way. To enter a literal Control-*a* (as used to go to the beginning of the line in Emacs or the Bash shell), use Control-*a a*.

Git for Scientists

In [Chapter 2](#), we discussed organizing a bioinformatics project directory and how this helps keep your work tidy during development. Good organization also facilitates automating tasks, which makes our lives easier and leads to more reproducible work. However, as our project changes over time and possibly incorporates the work of our collaborators, we face an additional challenge: managing different file versions.

It's likely that you already use some sort of versioning system in your work. For example, you may have files with names such as *thesis-vers1.docx*, *thesis-vers3_CD_edits.docx*, *analysis-vers6.R*, and *thesis-vers8_CD+GM+SW_edits.docx*. Storing these past versions is helpful because it allows us to go back and restore whole files or sections if we need to. File versions also help us differentiate our copies of a file from those edited by a collaborator. However, this ad hoc file versioning system doesn't scale well to complicated bioinformatics projects—our otherwise tidy project directories would be muddled with different versioned scripts, R analyses, *README* files, and papers.

Project organization only gets more complicated when we work collaboratively. We could share our entire directory with a colleague through a service like Dropbox or Google Drive, but we run the risk of something getting deleted or corrupted. It's also not possible to drop an entire bioinformatics project directory into a shared directory, as it likely contains gigabytes (or more) of data that may be too large to share across a network. These tools are useful for sharing small files, but aren't intended to manage large collaborative projects involving changing code and data.

Luckily, software engineers have faced these same issues in modern collaborative software development and developed *version control systems* (VCS) to manage different versions of collaboratively edited code. The VCS we'll use in this chapter was written by Linus Torvalds and is called *Git*. Linus wrote Git to manage the Linux kernel (which he also wrote), a large codebase with thousands of collaborators simultane-

ously changing and working on files. As you can imagine, Git is well suited for project version control and collaborative work.

Admittedly, Git can be tricky to learn at first. I highly recommend you take the time to learn Git in this chapter, but be aware that understanding Git (like most topics in this book, and arguably everything in life) will take time and practice. Throughout this chapter, I will indicate when certain sections are especially advanced; you can revisit these later without problems in continuity with the rest of the book. Also, I recommend you practice Git with the example projects and code from the book to get the basic commands in the back of your head. After struggling in the beginning with Git, you'll soon see how it's the best version control system out there.

Why Git Is Necessary in Bioinformatics Projects

As a longtime proponent of Git, I've suggested it to many colleagues and offered to teach them the basics. In most cases, I find the hardest part is actually in convincing scientists they should adopt version control in their work. Because you may be wondering whether working through this chapter is worth it, I want to discuss why learning Git is definitely worth the effort. If you're already 100% convinced, you can dive into learning Git in the next section.

Git Allows You to Keep Snapshots of Your Project

With version control systems, you create snapshots of your current project at specific points in its development. If anything goes awry, you can rewind to a past snapshot of your project's state (called a commit) and restore files. In the fast pace of bioinformatics work, having this safeguard is very useful.

Git also helps fix a frustrating type of bug known as software regression, where a piece of code that was once working mysteriously stops working or gives different results. For example, suppose that you're working on an analysis of SNP data. You find in your analysis that 14% of your SNPs fall in coding regions in one stretch of a chromosome. This is relevant to your project, so you cite this percent in your paper and make a commit.

Two months later, you've forgotten the details of this analysis, but need to revisit the 14% statistic. Much to your surprise, when you rerun the analysis code, this changes to 26%! If you've been tracking your project's development by making commits (e.g., taking snapshots), you'll have an entire history of all of your project's changes and can pinpoint when your results changed.

Git commits allow you to easily reproduce and rollback to past versions of analysis. It's also easy to look at every commit, when it was committed, what has changed across commits, and even compare the difference between any two commits. Instead

of redoing months of work to find a bug, Git can give you line-by-line code differences across versions.

In addition to simplifying bug finding, Git is an essential part of proper documentation. When your code produces results, it's essential that this version of code is fully documented for reproducibility. A good analogy comes from my friend and colleague Mike Covington: imagine you keep a lab notebook in pencil, and each time you run a new PCR you erase your past results and jot down the newest ones. This may sound extreme, but is functionally no different than changing code and not keeping a record of past versions.

Git Helps You Keep Track of Important Changes to Code

Most software changes over time as new features are added or bugs are fixed. It's important in scientific computing to follow the development of software we use, as a fixed bug could mean the difference between correct and incorrect results in our own work. Git can be very helpful in helping you track changes in code—to see this, let's look at a situation I've run into (and I suspect happens in labs all over the world).

Suppose a lab has a clever bioinformatician who has written a script that trims poor quality regions from reads. This bioinformatician then distributes this to all members of his lab. Two members of his lab send it to friends in other labs. About a month later, the clever bioinformatician realizes there's a bug that leads to incorrect results in certain cases. The bioinformatician quickly emails everyone in his lab the new version and warns them of the potential for incorrect results. Unfortunately, members of the other lab may not get the message and could continue using the older buggy version of the script.

Git helps solve this problem by making it easy to stay up to date with software development. With Git, it's easy to both track software changes and download new software versions. Furthermore, services like GitHub and Bitbucket host Git repositories on the Web, which makes sharing and collaborating on code across labs easy.

Git Helps Keep Software Organized and Available After People Leave

Imagine another situation: a postdoc moves to start her own lab, and all of her different software tools and scripts are scattered in different directories, or worse, completely lost. Disorderedly code disrupts and inconveniences other lab members; lost code leads to irreproducible results and could delay future research.

Git helps maintain both continuity in work and a full record of a project's history. Centralizing an entire project into a repository keeps it organized. Git stores every committed change, so the entire history of a project is available even if the main developer leaves and isn't around for questions. With the ability to roll back to past versions, modifying projects is less risky, making it easier to build off existing work.

Installing Git

If you're on OS X, install Git through Homebrew (e.g., `brew install git`); on Linux, use `apt-get` (e.g., `apt-get install git`). If your system does not have a package manager, [the Git website](#) has both source code and executable versions of Git.

Basic Git: Creating Repositories, Tracking Files, and Staging and Committing Changes

Now that we've seen some Git concepts and how Git fits into your bioinformatics workflow, let's explore the most basic Git concepts of creating repositories, telling Git which files to track, and staging and committing changes.

Git Setup: Telling Git Who You Are

Because Git is meant to help with collaborative editing of files, you need to tell Git who you are and what your email address is. To do this, use:

```
$ git config --global user.name "Sewall Wright"
$ git config --global user.email "swright@adaptivelandscape.org"
```

Make sure to use your own name and email, or course. We interact with Git through *subcommands*, which are in the format `git <subcommand>`. Git has loads of subcommands, but you'll only need a few in your daily work.

Another useful Git setting to enable now is terminal colors. Many of Git's subcommands use terminal colors to visually indicate changes (e.g., red for deletion and green for something new or modified). We can enable this with:

```
$ git config --global color.ui true
```

git init and git clone: Creating Repositories

To get started with Git, we first need to initialize a directory as a Git *repository*. A *repository* is a directory that's under version control. It contains both your current working files and snapshots of the project at certain points in time. In version control lingo, these snapshots are known as *commits*. Working with Git is fundamentally about creating and manipulating these commits: creating commits, looking at past commits, sharing commits, and comparing different commits.

With Git, there are two primary ways to create a repository: by initializing one from an existing directory, or cloning a repository that exists elsewhere. Either way, the result is a directory that Git treats as a repository. Git only manages the files and subdirectories inside the repository directory—it cannot manage files outside of your repository.

Let's start by initializing the *zmays-snps/* project directory we created in [Chapter 2](#) as a Git repository. Change into the *zmays-snps/* directory and use the Git subcommand `git init`:

```
$ git init
Initialized empty Git repository in /Users/vinceb/Projects/zmays-snps/.git/
```

`git init` creates a hidden directory called `.git/` in your *zmays-snps/* project directory (you can see it with `ls -a`). This `.git/` directory is how Git manages your repository behind the scenes. However, don't modify or remove anything in this directory—it's meant to be manipulated by Git only. Instead, we interact with our repository through Git subcommands like `git init`.

The other way to create a repository is through cloning an existing repository. You can clone repositories from anywhere: somewhere else on your filesystem, from your local network, or across the Internet. Nowadays, with repository hosting services like GitHub and Bitbucket, it's most common to clone Git repositories from the Web.

Let's practice cloning a repository from GitHub. For this example, we'll clone the Seqtk code from Heng Li's GitHub page. Seqtk is short for SEquence ToolKit, and contains a well-written and useful set of tools for processing FASTQ and FASTA files. First, [visit the GitHub repository](#) and poke around a bit. All of GitHub's repositories have this URL syntax: *user/repository*. Note on this repository's page that clone URL on the righthand side—this is where you can copy the link to clone this repository.

Now, let's switch to a directory outside of *zmays-snps/*. Whichever directory you choose is fine; I use a `~/src/` directory for cloning and compiling other developers' tools. From this directory, run:

```
$ git clone git://github.com/lh3/seqtk.git
Cloning into 'seqtk'...
remote: Counting objects: 92, done.
remote: Compressing objects: 100% (47/47), done.
remote: Total 92 (delta 56), reused 80 (delta 44)
Receiving objects: 100% (92/92), 32.58 KiB, done.
Resolving deltas: 100% (56/56), done.
```

`git clone` clones seqtk to your local directory, mirroring the original repository on GitHub. Note that you won't be able to directly modify Heng Li's original GitHub repository—cloning this repository only gives you access to retrieve new updates from the GitHub repository as they're released.

Now, if you `cd` into *seqtk/* and run `ls`, you'll see seqtk's source:

```
$ cd seqtk
$ ls
Makefile  README.md khash.h   kseq.h    seqtk.c
```

Despite originating through different methods, both *zmays-snps/* and *seqtk/* are Git repositories.

Tracking Files in Git: git add and git status Part I

Although you’ve initialized the *zmays-snps/* as a Git repository, Git doesn’t automatically begin tracking every file in this directory. Rather, you need to tell Git which files to *track* using the subcommand `git add`. This is actually a useful feature of Git—bioinformatics projects contain many files we don’t want to track, including large data files, intermediate results, or anything that could be easily regenerated by rerunning a command.

Before tracking a file, let’s use the command `git status` to check Git’s status of the files in our repository (switch to the *zmays-snps/* directory if you are elsewhere):

```
$ git status
# On branch master ❶
#
# Initial commit
#
# Untracked files: ❷
#   (use "git add <file>..." to include in what will be committed)
#
#       README
#       data/
nothing added to commit but untracked files present (use "git add" to track)
```

`git status` tell us:

- ❶ We’re on branch *master*, which is the default Git *branch*. Branches allow you to work on and switch between different versions of your project simultaneously. Git’s simple and powerful branches are a primary reason it’s such a popular version control system. We’re only going to work with Git’s default *master* branch for now, but we’ll learn more about branches later in this chapter.
- ❷ We have a list of “Untracked files,” which include everything in the root project directory. Because we haven’t told Git to track anything, Git has nothing to put in a commit if we were to try.

It’s good to get `git status` under your fingers, as it’s one of the most frequently used Git commands. `git status` describes the current state of your project repository: what’s changed, what’s ready to be included in the next commit, and what’s not being tracked. We’ll use it extensively throughout the rest of this chapter.

Let’s use `git add` to tell Git to track the *README* and *data/README* files in our *zmays-snps/* directory:

```
$ git add README data/README
```

Now, Git is tracking both *data/README* and *README*. We can verify this by running `git status` again:

```
$ ls
README  analysis data      scripts
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   README ❶
#       new file:   data/README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       data/seqs/ ❷
```

- ❶ Note now how Git lists *README* and *data/README* as new files, under the section “changes to be committed.” If we made a commit now, our commit would take a snapshot of the exact version of these files as they were when we added them with `git add`.
- ❷ There are also untracked directories like *data/seqs/*, as we have not told Git to track these yet. Conveniently, `git status` reminds us we could use `git add` to add these to a commit.

The *scripts/* and *analysis/* directories are not included in `git status` because they are empty. The *data/seqs/* directory is included because it contains the empty sequence files we created with `touch` in [Chapter 2](#).

Staging Files in Git: `git add` and `git status` Part II

With Git, there’s a difference between tracked files and files *staged* to be included in the next commit. This is a subtle difference, and one that often causes a lot of confusion for beginners learning Git. A file that’s tracked means Git knows about it. A staged file is not only tracked, but its latest changes are staged to be included in the next commit (see [Figure 5-1](#)).

A good way to illustrate the difference is to consider what happens when we change one of the files we started tracking with `git add`. Changes made to a tracked file will not automatically be included in the next commit. To include these new changes, we would need to explicitly *stage* them—using `git add` again. Part of the confusion lies

in the fact that `git add` both tracks new files and stages the changes made to tracked files. Let's work through an example to make this clearer.

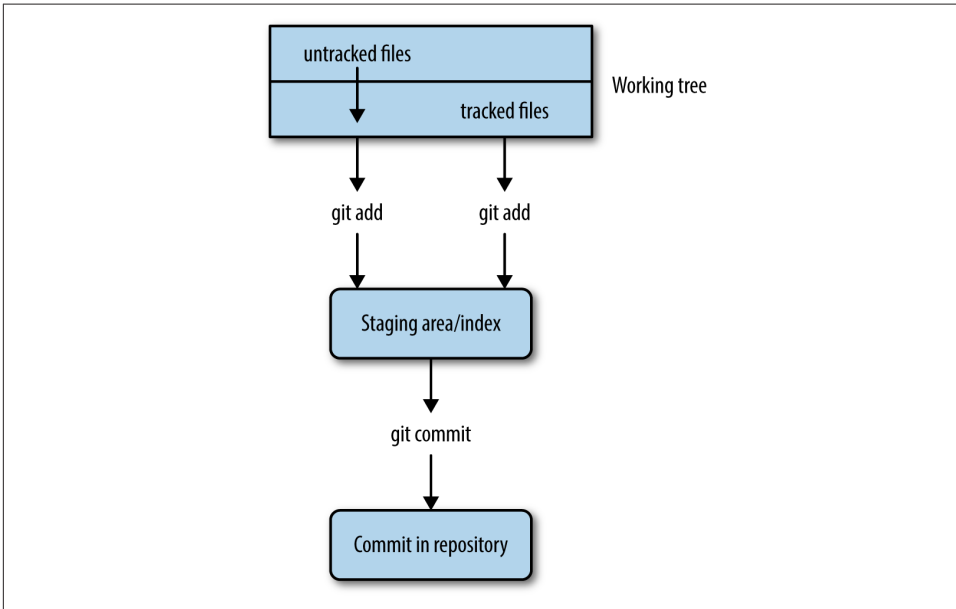


Figure 5-1. Git's separation of the working tree (all files in your repository), the staging area (files to be included in the next commit), and committed changes (a snapshot of a version of your project at some point in time); `git add` on an untracked file begins tracking it and stages it, while `git add` on a tracked file just stages it for the next commit

From the `git status` output from the last section, we see that both the `data/README` and `README` files are ready to be committed. However, look what happens when we make a change to one of these tracked files and then call `git status`:

```
$ echo "Zea Mays SNP Calling Project" >> README    # change file README
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   README
#       new file:   data/README
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
```

```
#      modified:   README
#
# Untracked files:
#  (use "git add <file>..." to include in what will be committed)
#
#      data/seqs/
```

After modifying *README*, `git status` lists *README* under “changes not staged for commit.” This is because we’ve made changes to this file since initially tracking and staging *README* with `git add` (when first tracking a file, its current version is also staged). If we were to make a commit now, our commit would include the previous version of *README*, *not* this newly modified version.

To add these recent modifications to *README* in our next commit, we stage them using `git add`. Let’s do this now and see what `git status` returns:

```
$ git add README
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   README
#       new file:   data/README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       data/seqs/
#       notebook.md
```

Now, *README* is listed under “Changes to be committed” again, because we’ve staged these changes with `git add`. Our next commit will include the most recent version.

Again, don’t fret if you find this confusing. The difference is subtle, and it doesn’t help that we use `git add` for both operations. Remember the two roles of `git add`:

- Alerting Git to start tracking untracked files (this also stages the current version of the file to be included in the next commit)
- Staging changes made to an already tracked file (staged changes will be included in the next commit)

It’s important to be aware that any modifications made to a file since the last time it was staged will *not* be included in the next commit unless they are staged with `git add`. This extra step may seem like an inconvenience but actually has many benefits.

Suppose you’ve made changes to many files in a project. Two of these files’ changes are complete, but everything else isn’t quite ready. Using Git’s staging, you can stage and commit only these two complete files and keep other incomplete files out of your commit. Through planned staging, your commits can reflect meaningful points in development rather than random snapshots of your entire project directory (which would likely include many files in a state of disarray). When we learn about committing in the next section, we’ll see a shortcut to stage and commit all modified files.

git commit: Taking a Snapshot of Your Project

We’ve spoken a lot about commits, but haven’t actually made one yet. When first learning Git, the trickiest part of making a commit is understanding staging. Actually committing your staged commits is quite easy:

```
$ git commit -m "initial import"
2 files changed, 1 insertion(+)
create mode 100644 README
create mode 100644 data/README
```

This command commits your staged changes to your repository with the *commit message* “initial import.” Commit messages are notes to your collaborators (and yourself in the future) about what a particular commit includes. Optionally, you can omit the `-m` option, and Git will open up your default text editor. If you prefer to write commit messages in a text editor (useful if they are multiline messages), you can change the default editor Git uses with:

```
$ git config --global core.editor emacs
```

where `emacs` can be replaced by `vim` (the default) or another text editor of your choice.



Some Advice on Commit Messages

Commit messages may seem like an inconvenience, but it pays off in the future to have a description of how a commit changes code and what functionality is affected. In three months when you need to figure out why your SNP calling analyses are returning unexpected results, it’s much easier to find relevant commits if they have messages like “modifying SNP frequency function to fix singleton bug, refactored coverage calculation” rather than “cont” (that’s an actual commit I’ve seen in a public project). For an entertaining take on this, see [xkcd’s “Git Commit” comic](#).

Earlier, we staged our changes using `git add`. Because programmers like shortcuts, there’s an easy way to stage all tracked files’ changes and commit them in one command: `git commit -a -m "your commit message"`. The option `-a` tells `git commit`

to automatically stage all modified tracked files in this commit. Note that while this saves time, it also will throw *all* changes to tracked files in this commit. Ideally commits should reflect helpful snapshots of your project’s development, so including every slightly changed file may later be confusing when you look at your repository’s history. Instead, make frequent commits that correspond to discrete changes to your project like “new counting feature added” or “fixed bug that led to incorrect translation.”

We’ve included all changes in our commit, so our working directory is now “clean”: no tracked files differ from the version in the last commit. Until we make modifications, `git status` indicates there’s nothing to commit:

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       data/seqs/
```

Untracked files and directories will still remain untracked (e.g., *data/seqs/*), and any unstaged changes to tracked files will not be included in the next commit unless added. Sometimes a working directory with unstaged changes is referred to as “messy,” but this isn’t a problem.

Seeing File Differences: `git diff`

So far we’ve seen the Git tools needed to help you stage and commit changes in your repository. We’ve used the `git status` subcommand to see which files are tracked, which have changes, and which are staged for the next commit. Another subcommand is quite helpful in this process: `git diff`.

Without any arguments, `git diff` shows you the difference between the files in your working directory and what’s been staged. If none of your changes have been staged, `git diff` shows us the difference between your last commit and the current versions of your files. For example, if I add a line to *README.md* and run `git diff`:

```
$ echo "Project started 2013-01-03" >> README
$ git diff
diff --git a/README b/README
index 5483cfd..ba8d7fc 100644
--- a/README ❶
+++ b/README
@@ -1,2 @@ ❷
   Zea Mays SNP Calling Project
+Project started 2013-01-03 ❸
```


This format (called a *unified diff*) may seem a bit cryptic at first. When Git’s terminal colors are enabled, `git diff`’s output is easier to read, as added lines will be green and deleted lines will be red.

- ❶ This line (and the one following it) indicate there are two versions of the *README* file we are comparing, a and b. The `---` indicates the original file—in our case, the one from our last commit. `+++` indicates the changed version.
- ❷ This denotes the start of a changed hunk (hunk is diff’s term for a large changed block), and indicates which line the changes start on, and how long they are. Diffs try to break your changes down into hunks so that you can easily identify the parts that have been changed. If you’re curious about the specifics, see [Wikipedia’s page on the diff utility](#).
- ❸ Here’s the meat of the change. Spaces before the line (e.g., the line that begins *Zea Mays...* indicates nothing was changed (and just provide context). Plus signs indicate a line addition (e.g., the line that begins *Project...*), and negative signs indicate a line deletion (not shown in this diff because we’ve only added a line). Changes to a line are represented as a deletion of the original line and an addition of the new line.

After we stage a file, `git diff` won’t show any changes, because `git diff` compares the version of files in your working directory to the last staged version. For example:

```
$ git add README
$ git diff # shows nothing
```

If we wanted to compare what’s been staged to our last commit (which will show us exactly what’s going into the next commit), we can use `git diff --staged` (in old versions of Git this won’t work, so upgrade if it doesn’t). Indeed, we can see the change we just staged:

```
$ git diff --staged
diff --git a/README b/README
index 5483cfd..ba8d7fc 100644
--- a/README
+++ b/README
@@ -1,2 @@
  Zea Mays SNP Calling Project
+Project started 2013-01-03
```

`git diff` can also be used to compare arbitrary objects in our Git commit history, a topic we’ll see in [“More git diff: Comparing Commits and Files” on page 100](#).

Seeing Your Commit History: git log

Commits are like chains (more technically, directed acyclic graphs), with each commit pointing to its parent (as in [Figure 5-2](#)).

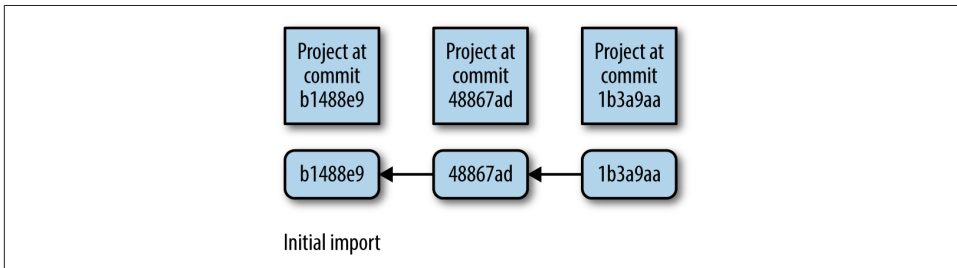


Figure 5-2. Commits in Git take discrete snapshots of your project at some point in time, and each commit (except the first) points to its parent commit; this chain of commits is your set of connected snapshots that show how your project repository evolves

We can use `git log` to visualize our chain of commits:

```
$ git log
commit 3d7ffa6f0276e607dcd94e18d26d21de2d96a460 ①
Author: Vince Buffalo <vsbuffaloAAAAA@gmail.com>
Date:   Mon Sep 23 23:55:08 2013 -0700

    initial import
```

- ① This strange looking mix of numbers and characters is a *SHA-1* checksum. Each commit will have one of these, and they will depend on your repository's past commit history and the current files. SHA-1 hashes act as a unique ID for each commit in your repository. You can always refer to a commit by its SHA-1 hash.



git log and Your Terminal Pager

`git log` opens up your repository's history in your default *pager* (usually either the program `more` or `less`). If you're unfamiliar with pagers, `less`, and `more`, don't fret. To exit and get back to your prompt, hit the letter `q`. You can move forward by pressing the space bar, and move backward by pressing `b`. We'll look at `less` in more detail in [Chapter 7](#).

Let's commit the change we made in the last section:

```
$ git commit -a -m "added information about project to README"
[master 94e2365] added information about project to README
1 file changed, 1 insertion(+)
```

Now, if we look at our commit history with `git log`, we see:

```
$ git log
commit 94e2365dd66701a35629d29173d640fdae32fa5c
Author: Vince Buffalo <vsbuffaloAAAAA@gmail.com>
Date: Tue Sep 24 00:02:11 2013 -0700
```

added information about project to README

```
commit 3d7ffa6f0276e607dcd94e18d26d21de2d96a460
Author: Vince Buffalo <vsbuffaloAAAAA@gmail.com>
Date: Mon Sep 23 23:55:08 2013 -0700
```

initial import

As we continue to make and commit changes to our repository, this chain of commits will grow. If you want to see a nice example of a longer Git history, change directories to the *seqtk* repository we cloned earlier and call `git log`.

Moving and Removing Files: `git mv` and `git rm`

When Git tracks your files, it wants to be in charge. Using the command `mv` to move a tracked file will confuse Git. The same applies when you remove a file with `rm`. To move or remove tracked files in Git, we need to use Git's version of `mv` and `rm`: `git mv` and `git rm`.

For example, our *README* file doesn't have an extension. This isn't a problem, but because the *README* file might later contain Markdown, it's not a bad idea to change its extension to *.md*. You can do this using `git mv`:

```
$ git mv README README.md
$ git mv data/README data/README.md
```

Like all changes, this isn't stored in your repository until you commit it. If you `ls` your files, you can see your working copy has been renamed:

```
$ ls
README.md  analysis  data      notebook.md  scripts
```

Using `git status`, we see this change is staged and ready to be committed:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    README -> README.md
#       renamed:    data/README -> data/README.md
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       data/seqs/
```

`git mv` already staged these commits for us; `git add` is only necessary for staging modifications to the contents of files, not moving or removing files. Let's commit these changes:

```
$ git commit -m "added markdown extensions to README files"
[master e4feb22] added markdown extensions to README files
2 files changed, 0 insertions(+), 0 deletions(-)
rename README => README.md (100%)
rename data/{README => README.md} (100%)
```

Note that even if you change or remove a file and commit it, it still exists in past snapshots. Git does its best to make everything recoverable. We'll see how to recover files later on in this chapter.

Telling Git What to Ignore: `.gitignore`

You may have noticed that `git status` keeps listing which files are not tracked. As the number of files in your bioinformatics project starts to increase (this happens quickly!) this long list will become a burden.

Many of the items in this untracked list may be files we never want to commit. Sequencing data files are a great example: they're usually much too large to include in a repository. If we were to commit these large files, collaborators cloning your repository would have to download these enormous data files. We'll talk about other ways of managing these later, but for now, let's just ignore them.

Suppose we wanted to ignore all FASTQ files (with the extension *fastq*) in the *data/seqs/* directory. To do this, create and edit the file *.gitignore* in your *zmays-snps/* repository directory, and add:

```
data/seqs/*.fastq
```

Now, `git status` gives us:

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       .gitignore
```

It seems we've gotten rid of one annoyance (the *data/seqs/* directory in "Untracked files") but added another (the new *.gitignore*). Actually, the best way to resolve this is to add and commit your *.gitignore* file. It may seem counterintuitive to contribute a file to a project that's merely there to tell Git what to ignore. However, this is a good practice; it saves collaborators from seeing a listing of untracked files Git should ignore. Let's go ahead and stage the *.gitignore* file, and commit this and the filename changes we made earlier:

```
$ git add .gitignore
$ git commit -m "added .gitignore"
[master c509f63] added .gitignore
1 file changed, 1 insertion(+)
create mode 100644 .gitignore
```

What should we tell *.gitignore* to ignore? In the context of a bioinformatics project, here are some guidelines:

Large files

These should be ignored and managed by other means, as Git isn't designed to manage really large files. Large files slow creating, pushing, and pulling commits. This can lead to quite a burden when collaborators clone your repository.

Intermediate files

Bioinformatics projects are often filled with intermediate files. For example, if you align reads to a genome, this will create SAM or BAM files. Even if these aren't large files, these should probably be ignored. If a data file can easily be reproduced by rerunning a command (or better yet, a script), it's usually preferable to just store how it was created. Ultimately, recording and storing how you created an intermediate file in Git is more important than the actual file. This also ensures reproducibility.

Text editor temporary files

Text editors like Emacs and Vim will sometimes create temporary files in your directory. These can look like *textfile.txt~* or *#textfile.txt#*. There's no point in storing these in Git, and they can be an annoyance when viewing progress with `git status`. These files should always be added to *.gitignore*. Luckily, *.gitignore* takes wildcards, so these can be ignored with entries like **~* and *\#*\#*.

Temporary code files

Some language interpreters (e.g., Python) produce temporary files (usually with some sort of optimized code). With Python, these look like *overlap.pyc*.

We can use a *global .gitignore* file to universally ignore a file across all of our projects. Good candidates of files to globally ignore are our text editor's temporary files or files your operating system creates (e.g., OS X will sometimes create hidden files named *.DS_Store* in directories to store details like icon position). GitHub maintains [a useful repository of global .gitignore suggestions](#).

You can create a global *.gitignore* file in *~/.gitignore_global* and then configure Git to use this with the following:

```
git config --global core.excludesfile ~/.gitignore_global
```

A repository should store everything required to replicate a project except large datasets and external programs. This includes all scripts, documentation, analysis, and possibly even a final manuscript. Organizing your repository this way means that all

of your project's *dependencies* are in one place and are managed by Git. In the long run, it's far easier to have Git keep track of your project's files, than try to keep track of them yourself.

Undoing a Stage: git reset

Recall that one nice feature of Git is that you don't have to include messy changes in a commit—just don't stage these files. If you accidentally stage a messy file for a commit with `git add`, you can unstage it with `git reset`. For example, suppose you add a change to a file, stage it, but decide it's not ready to be committed:

```
$ echo "TODO: ask sequencing center about adapters" >> README.md
$ git add README.md
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README.md
#
```

With `git status`, we can see that our change to *README.md* is going to be included in the next commit. To unstage this change, follow the directions `git status` provides:

```
$ git reset HEAD README.md
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
directory)
#
#       modified:   README.md
#
```

The syntax seems a little strange, but all we're doing is resetting our staging area (which Git calls the *index*) to the version at *HEAD* for our *README.md* file. In Git's lingo, *HEAD* is an alias or pointer to the last commit on the *current* branch (which is, as mentioned earlier, the default Git branch called *master*). Git's `reset` command is a powerful tool, but its default action is to just reset your index. We'll see additional ways to use `git reset` when we learn about working with commit histories.

Collaborating with Git: Git Remotes, git push, and git pull

Thus far, we've covered the very basics of Git: tracking and working with files, staging changes, making commits, and looking at our commit history. Commits are the foundation of Git—they are the snapshots of our project as it evolves. Commits allow you

to go back in time and look at, compare, and recover past versions, which are all topics we look at later in this chapter. In this section, we're going to learn how to collaborate with Git, which at its core is just about sharing commits between your repository and your collaborators' repositories.

The basis of sharing commits in Git is the idea of a *remote repository*, which is just a version of your repository hosted elsewhere. This could be a shared departmental server, your colleague's version of your repository, or on a repository hosting service like GitHub or Bitbucket. Collaborating with Git first requires we configure our local repository to work with our remote repositories. Then, we can retrieve commits from a remote repository (a *pull*) and send commits to a remote repository (a *push*).

Note that Git, as a *distributed* version control system, allows you to work with remote repositories any way you like. These *workflow* choices are up to you and your collaborators. In this chapter, we'll learn an easy common workflow to get started with: collaborating over a *shared central repository*.

Let's take a look at an example: suppose that you're working on a project you wish to share with a colleague. You start the project in your local repository. After you've made a few commits, you want to share your progress by sharing these commits with your collaborator. Let's step through the entire workflow before seeing how to execute it with Git:

1. You create a shared central repository on a server that both you and your collaborator have access to.
2. You push your project's initial commits to this repository (seen in (a) in [Figure 5-3](#)).
3. Your collaborator then retrieves your initial work by cloning this central repository (seen in (b) in [Figure 5-3](#)).
4. Then, your collaborator makes her changes to the project, commits them to her local repository, and then pushes these commits to the central repository (seen in (a) in [Figure 5-4](#)).
5. You then pull in the commits your collaborator pushed to the central repository (seen in (b) in [Figure 5-4](#)). The commit history of your project will be a mix of both you and your collaborator's commits.

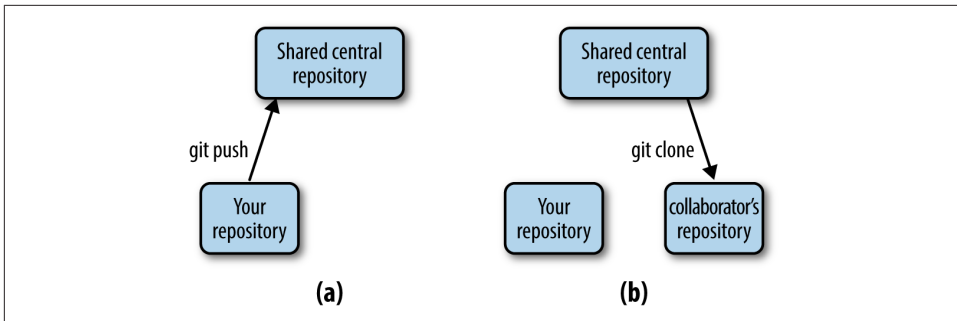


Figure 5-3. After creating a new shared central repository, you push your project's commits (a); your collaborator can retrieve your project and its commits by cloning this central repository (b)

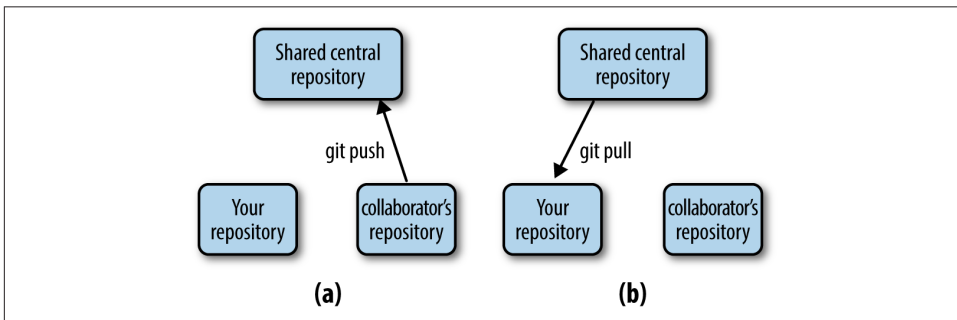


Figure 5-4. After making and committing changes, your collaborator pushes them to the central repository (a); to retrieve your collaborator's new commits, you pull them from the central repository (b)

This process then repeats: you and your collaborator work independently in your own local repositories, and when either of you have commits to share, you push them to the central repository. In many cases, if you and your collaborator work on different files or different sections of the same file, Git can automatically figure out how best to merge these changes. This is an amazing feature of collaborating with Git: you and your collaborator can work on the same project simultaneously. Before we dive into how to do this, there is one caveat to discuss.

It's important to note that Git cannot always automatically merge code and documents. If you and your collaborator are both editing the same section of the same file and both of you commit these changes, this will create a *merge conflict*. Unfortunately, one of you will have to resolve the conflicting files manually. Merge conflicts occur when you (or your collaborator) pull in commits from the central repository and your collaborator's (or your) commits conflict with those in the local repository. In

these cases, Git just isn't smart enough to figure out how to reconcile you and your collaborator's conflicting versions.

Most of us are familiar with this process when we collaboratively edit manuscripts in a word processor. If you write a manuscript and send it to all your collaborators to edit, you will need to manually settle sections with conflicting edits. Usually, we get around this messy situation through planning and prioritizing which coauthors will edit first, and gradually incorporating changes. Likewise, good communication and planning with your collaborators can go far in preventing Git merge conflicts, too. Additionally, it's helpful to frequently push and pull commits to the central repository; this keeps all collaborators synced so everyone's working with the newest versions of files.

Creating a Shared Central Repository with GitHub

The first step of our workflow is to create a shared central repository, which is what you and your collaborator(s) share commits through. In our examples, we will use GitHub, a web-based Git repository hosting service. **Bitbucket** is another Git repository hosting service you and your collaborators could use. Both are excellent; we'll use GitHub because it's already home to many large bioinformatics projects like Biopython and Samtools.

Navigate to <http://github.com> and sign up for an account. After your account is set up, you'll be brought to the GitHub home page, which is a newsfeed for you to follow project development (this newsfeed is useful to follow how bioinformatics software you use changes over time). On the main page, there's a link to create a new repository. After you've navigated to **the Create a New Repository page**, you'll see you need to provide a repository name, and you'll have the choice to initialize with a *README.md* file (GitHub plays well with Markdown), a *.gitignore* file, and a license (to license your software project). For now, just create a repository named *zmays-snps*. After you've clicked the "Create repository" button, GitHub will forward you to an empty repository page—the public frontend of your project.

There are a few things to note about GitHub:

- Public repositories are free, but private repositories require you to pay. Luckily, GitHub has **a special program for educational users**. If you need private repositories without cost, Bitbucket has a different pricing scheme and provides some for free. Or, you can set up your own internal Git repository on your network if you have shared server space. Setting up your own Git server is out of the scope of this book, but see "Git on the Server - Setting Up the Server" in Scott Chacon and Ben Straub's free online book *Pro Git* for more information. If your repository is public, anyone can see the source (and even clone and develop their own ver-

sions of your repository). However, other users don't have access to modify your GitHub repository unless you grant it to them.

- If you're going to use GitHub for collaboration, all participating collaborators need a GitHub account.
- By default, you are the only person who has write (push) access to the repository you created. To use your remote repository as a shared central repository, you'll have to add *collaborators* in your GitHub repository's settings. Collaborators are GitHub users who have access to push their changes to your repository on GitHub (which modifies it).
- There are other common GitHub workflows. For example, if you manage a lab or other group, you can set up an organization account. You can create repositories and share them with collaborators under the organization's name. We'll discuss other GitHub workflows later in this chapter.

Authenticating with Git Remotes

GitHub uses SSH keys to authenticate you (the same sort we generated in [“Quick Authentication with SSH Keys” on page 59](#)). SSH keys prevent you from having to enter a password each time you push or pull from your remote repository. Recall in [“Quick Authentication with SSH Keys” on page 59](#) we generated two SSH keys: a public key and a private key. Navigate to your account settings on GitHub, and in account settings, find the SSH keys tab. Here, you can enter your public SSH key (remember, don't share your private key!) by using `cat ~/.ssh/id_rsa.pub` to view it, copying it to your clipboard, and pasting it into GitHub's form. You can then try out your SSH public key by using:

```
$ ssh -T git@github.com
Hi vsbuffalo! You've successfully authenticated, but
GitHub does not provide shell access.
```

If you're having troubles with this, consult GitHub's [“Generating SSH Keys” article](#).

GitHub allows you to use HTTP as a protocol, but this is typically only used if your network blocks SSH. By default, HTTP asks for passwords each time you try to pull and push (which gets tiresome quickly), but there are ways around this—see GitHub's [“Caching Your GitHub Password in Git” article](#).

Connecting with Git Remotes: `git remote`

Now, let's configure our local repository to use the GitHub repository we've just created as a remote repository. We can do this with `git remote add`:

```
$ git remote add origin git@github.com:username/zmays-snps.git
```

In this command, we specify not only the address of our Git repository (*git@github.com:username/zmays-snps.git*), but also a name for it: *origin*. By convention, *origin* is the name of your primary remote repository. In fact, earlier when we cloned Seqtk from GitHub, Git automatically added the URL we cloned from as a remote named *origin*.

Now if you enter `git remote -v` (the `-v` makes it more verbose), you see that our local Git repository knows about the remote repository:

```
$ git remote -v
origin  git@github.com:username/zmays-snps.git (fetch)
origin  git@github.com:username/zmays-snps.git (push)
```

Indeed, *origin* is now a repository we can push commits to and fetch commits from. We'll see how to do both of these operations in the next two sections.

It's worth noting too that you can have multiple remote repositories. Earlier, we mentioned that Git is a distributed version control system; as a result, we can have many remote repositories. We'll come back to how this is useful later on. For now, note that you can add other remote repositories with different names. If you ever need to delete an unused remote repository, you can with `git remote rm <repository-name>`.

Pushing Commits to a Remote Repository with `git push`

With our remotes added, we're ready to share our work by pushing our commits to a remote repository. Collaboration on Git is characterized by repeatedly pushing your work to allow your collaborators to see and work on it, and pulling their changes into your own local repository. As you start collaborating, remember you only share the commits you've made.

Let's push our initial commits from *zmays-snps* into our remote repository on GitHub. The subcommand we use here is `git push <remote-name> <branch>`. We'll talk more about using branches later, but recall from [“Tracking Files in Git: git add and git status Part I” on page 72](#) that our default branch name is *master*. Thus, to push our *zmays-snps* repository's commits, we do this:

```
$ git push origin master
Counting objects: 14, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (14/14), 1.24 KiB | 0 bytes/s, done.
Total 14 (delta 0), reused 0 (delta 0)
To git@github.com:vsbuffalo/zmays-snps.git
 * [new branch]      master -> master
```

That's it—your collaborator now has access to all commits that were on your *master* branch through the central repository. Your collaborator retrieves these commits by pulling them from the central repository into her own local repository.

Pulling Commits from a Remote Repository with `git pull`

As you push new commits to the central repository, your collaborator's repository will go out of date, as there are commits on the shared repository she doesn't have in her own local repository. She'll need to pull these commits in before continuing with her work. Collaboration on Git is a back-and-forth exchange, where one person pushes their latest commits to the remote repository, other collaborators pull changes into their local repositories, make their own changes and commits, and then push these commits to the central repository for others to see and work with.

To work through an example of this exchange, we will clone our own repository to a different directory, mimicking a collaborator's version of the project. Let's first clone our remote repository to a local directory named *zmays-snps-barbara/*. This directory name reflects that this local repository is meant to represent our colleague Barbara's repository. We can clone *zmays-snps* from GitHub to a local directory named *zmays-snps-barbara/* as follows:

```
$ git clone git@github.com:vsbuffalo/zmays-snps.git zmays-snps-barbara
Cloning into 'zmays-snps-barbara'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 14 (delta 0), reused 14 (delta 0)
Receiving objects: 100% (14/14), done.
Checking connectivity... done
```

Now, both repositories have the same commits. You can verify this by using `git log` and seeing that both have the same commits. Now, in our original *zmays-snps/* local repository, let's modify a file, make a commit, and push to the central repository:

```
$ echo "Samples expected from sequencing core 2013-01-10" >> README.md
$ git commit -a -m "added information about samples"
[master 46f0781] added information about samples
1 file changed, 1 insertion(+)
$ git push origin master
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 415 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:vsbuffalo/zmays-snps.git
c509f63..46f0781 master -> master
```

Now, Barbara's repository (*zmays-snps-barbara*) is a commit behind both our local *zmays-snps* repository and the central shared repository. Barbara can pull in this change as follows:

```
$ # in zmays-snps-barbara/
$ git pull origin master
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
```

```

remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From github.com:vsbuffalo/zmays-snps
* branch          master      -> FETCH_HEAD
c509f63..46f0781  master      -> origin/master
Updating c509f63..46f0781
Fast-forward
 README.md | 1 +
1 file changed, 1 insertion(+)

```

We can verify that Barbara's repository contains the most recent commit using `git log`. Because we just want a quick image of the last few commits, I will use `git log` with some helpful formatting options:

```

$ # in zmays-snps-barbara/
$ git log --pretty=oneline --abbrev-commit
46f0781 added information about samples
c509f63 added .gitignore
e4feb22 added markdown extensions to README files
94e2365 added information about project to README
3d7ffa6 initial import

```

Now, our commits are in both the central repository and Barbara's repository.

Working with Your Collaborators: Pushing and Pulling

Once you grow a bit more acquainted with pushing and pulling commits, it will become second nature. I recommend practicing this with fake repositories with a lab-mate or friend to get the hang of it. Other than merge conflicts (which we cover in the next section), there's nothing tricky about pushing and pulling. Let's go through a few more pushes and pulls so it's extremely clear.

In the last section, Barbara pulled our new commit into her repository. But she will also create and push her own commits to the central repository. To continue our example, let's make a commit from Barbara's local repository and push it to the central repository. Because there is no Barbara (Git is using the account we made at the beginning of this chapter to make commits), I will modify `git log`'s output to show Barbara as the collaborator. Suppose she adds the following line to the *README.md*:

```

$ # in zmays-snps-barbara/ -- Barbara's version
$ echo "\n\nMaize reference genome version: refgen3" >> README.md
$ git commit -a -m "added reference genome info"
[master 269aa09] added reference genome info
1 file changed, 3 insertions(+)
$ git push origin master
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 390 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)

```

```
To git@github.com:vsbuffalo/zmays-snps.git
46f0781..269aa09  master -> master
```

Now Barbara's local repository and the central repository are two commits ahead of our local repository. Let's switch to our *zmays-snps* repository, and pull these new commits in. We can see how Barbara's commits changed *README.md* with `cat`:

```
$ # in zmays-snps/ -- our version
$ git pull origin master
From github.com:vsbuffalo/zmays-snps
* branch      master      -> FETCH_HEAD
Updating 46f0781..269aa09
Fast-forward
 README.md | 3 +++
 1 file changed, 3 insertions(+)
```

```
$ cat README.md
Zea Mays SNP Calling Project
Project started 2013-01-03
Samples expected from sequencing core 2013-01-10
```

Maize reference genome version: refgen3

If we were to look at the last two log entries, they would look as follows:

```
$ git log -n 2
commit 269aa09418b0d47645c5d077369686ff04b16393
Author: Barbara <barbara@barbarasmaize.com>
Date:   Sat Sep 28 22:58:55 2013 -0700

    added reference genome info

commit 46f0781e9e081c6c9ee08b2d83a8464e9a26ae1f
Author: Vince Buffalo <vsbuffaloAAAAA@gmail.com>
Date:   Tue Sep 24 00:31:31 2013 -0700

    added information about samples
```

This is what collaboration looks like in Git's history: a set of sequential commits made by different people. Each is a snapshot of their repository and the changes they made since the last commit. All commits, whether they originate from your collaborator's or your repository, are part of the same history and point to their parent commit.

Because new commits build on top of the commit history, it's helpful to do the following to avoid problems:

- When pulling in changes, it helps to have your project's changes committed. Git will error out if a pull would change a file that you have uncommitted changes to, but it's still helpful to commit your important changes before pulling.

- Pull often. This complements the earlier advice: planning and communicating what you'll work on with your collaborators. By pulling in your collaborator's changes often, you're in a better position to build on your collaborators' changes. Avoid working on older, out-of-date commits.

Merge Conflicts

Occasionally, you'll pull in commits and Git will warn you there's a merge conflict. Resolving merge conflicts can be a bit tricky—if you're struggling with this chapter so far, you can bookmark this section and return to it when you encounter a merge conflict in your own work.

Merge conflicts occur when Git can't automatically merge your repository with the commits from the latest pull—Git needs your input on how best to resolve a conflict in the version of the file. Merge conflicts seem scary, but the strategy to solve them is always the same:

1. Use `git status` to find the conflicting file(s).
2. Open and edit those files manually to a version that fixes the conflict.
3. Use `git add` to tell Git that you've resolved the conflict in a particular file.
4. Once all conflicts are resolved, use `git status` to check that all changes are staged. Then, commit the resolved versions of the conflicting file(s). It's also wise to immediately push this merge commit, so your collaborators see that you've resolved a conflict and can continue their work on this new version accordingly.

As an example, let's create a merge conflict between our *zmays-snps* repository and Barbara's *zmays-snps-barbara* repository. One common situation where merge conflicts arise is to pull in a collaborator's changes that affect a file you've made and committed changes to. For example, suppose that Barbara changed *README.md* to something like the following (you'll have to do this in your text editor if you're following along):

```
Zea Mays SNP Calling Project
Project started 2013-01-03
Samples expected from sequencing core 2013-01-10
```

```
Maize reference genome version: refgen3, downloaded 2013-01-04 from
http://maizegdb.org into `/share/data/refgen3/`.
```

After making these edits to *README.md*, Barbara commits and pushes these changes. Meanwhile, in your repository, you also changed the last line:

```
Zea Mays SNP Calling Project
Project started 2013-01-03
```

Samples expected from sequencing core 2013-01-10

We downloaded refgen3 on 2013-01-04.

You commit this change, and then try to push to the shared central repository. To your surprise, you get the following error message:

```
$ git push origin master
To git@github.com:vsbuffalo/zmays-snps.git
! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'git@github.com:vsbuffalo/zmays-snps.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Git rejects your push attempt because Barbara has already updated the central repository's *master* branch. As Git's message describes, we need to resolve this by integrating the commits Barbara has pushed into our own local repository. Let's pull in Barbara's commit, and then try pushing as the message suggests (note that this error is not a merge conflict—rather, it just tells us we can't push to a remote that's one or more commits ahead of our local repository):

```
$ git pull origin master
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1)
Unpacking objects: 100% (3/3), done.
From github.com:vsbuffalo/zmays-snps
* branch            master      -> FETCH_HEAD
   269aa09..dafce75  master      -> origin/master
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

This is the merge conflict. This message isn't very helpful, so we follow the first step of the merge strategy by checking everything with `git status`:

```
$ git status
# On branch master
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
#       both modified:   README.md
#
no changes added to commit (use "git add" and/or "git commit -a")
```


`git status` tells us that there is only one file with a merge conflict, *README.md* (because we both edited it). The second step of our strategy is to look at our conflicting file(s) in our text editor:

```
Zea Mays SNP Calling Project
Project started 2013-01-03
Samples expected from sequencing core 2013-01-10

<<<<<< HEAD ❶
We downloaded refgen3 on 2013-01-04.
===== ❷
Maize reference genome version: refgen3, downloaded 2013-01-04 from
http://maizegdb.org into `/share/data/refgen3/`.
>>>>>> dafce75dc531d123922741613d8f29b894e605ac ❸
```

Notice Git has changed the content of this file in indicating the conflicting lines.

- ❶ This is the start of our version, the one that's HEAD in our repository. HEAD is Git's lingo for the latest commit (technically, HEAD points to the latest commit on the current branch).
- ❷ Indicates the end of HEAD and beginning of our collaborator's changes.
- ❸ This final delimiter indicates the end of our collaborator's version, and the different conflicting chunk. Git does its best to try to isolate the conflicting lines, so there can be many of these chunks.

Now we use step two of our merge conflict strategy: edit the conflicting file to resolve all conflicts. Remember, Git raises merge conflicts when it can't figure out what to do, so you're the one who has to manually resolve the issue. Resolving merge conflicts in files takes some practice. After resolving the conflict in *README.md*, the edited file would appear as follows:

```
Zea Mays SNP Calling Project
Project started 2013-01-03
Samples expected from sequencing core 2013-01-10

We downloaded the B73 reference genome (refgen3) on 2013-01-04 from
http://maizegdb.org into `/share/data/refgen3/`.
```

I've edited this file so it's a combination of both versions. We're happy now with our changes, so we continue to the third step of our strategy—using `git add` to declare this conflict has been resolved:

```
$ git add README.md
```

Now, the final step in our strategy—check `git status` to ensure all conflicts are resolved and ready to be merged, and commit them:

```
$ git status
git status
# On branch master
# All conflicts fixed but you are still merging.
# (use "git commit" to conclude merge)
#
# Changes to be committed:
#
#       modified:   README.md
#

$ git commit -a -m "resolved merge conflict in README.md"
[master 20041ab] resolved merge conflict in README.md
```

That's it: our merge conflict is resolved! With our local repository up to date, our last step is to share our merge commit with our collaborator. This way, our collaborators know of the merge and can continue their work from the new merged version of the file.

After pushing our merge commit to the central repository with `git push`, let's switch to Barbara's local repository and pull in the merge commit:

```
$ git pull origin master
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 2), reused 5 (delta 2)
Unpacking objects: 100% (6/6), done.
From github.com:vsbuffalo/zmays-snps
 * branch          master      -> FETCH_HEAD
   dafce75..20041ab master      -> origin/master
Updating dafce75..20041ab
Fast-forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Using `git log` we see that this is a special commit—a merge commit:

```
commit cd72acf0a81cdd688cb713465cb774320caeb2fd
Merge: f9114a1 d99121e
Author: Vince Buffalo <vsbuffaloAAAAA@gmail.com>
Date:   Sat Sep 28 20:38:01 2013 -0700

    resolved merge conflict in README.md
```

Merge commits are special, in that they have two parents. This happened because both Barbara and I committed changes to the same file with the same parent commit. Graphically, this situation looks like [Figure 5-5](#).

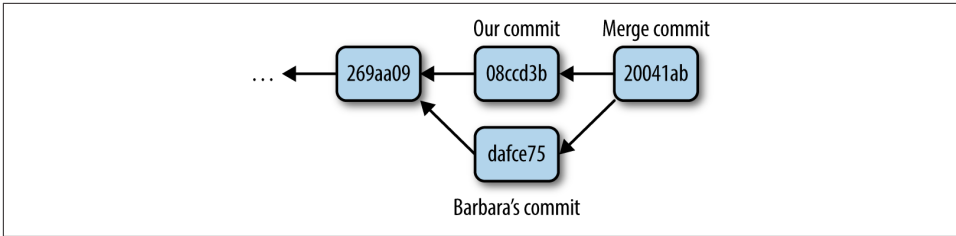


Figure 5-5. A merge commit has two parents—in this case, Barbara’s version and our version; merge commits resolve conflicts between versions

We can also see the same story through `git log` with the option `--graph`, which draws a plain-text graph of your commits:

```

*   commit 20041abaab156c39152a632ea7e306540f89f706
| \ Merge: 08ccd3b dafce75
|  | Author: Vince Buffalo <vsbuffaloAAAAAA@gmail.com>
|  | Date:   Sat Sep 28 23:13:07 2013 -0700
|  |
|  |     resolved merge conflict in README.md
|  |
|  | * commit dafce75dc531d123922741613d8f29b894e605ac
|  | | Author: Vince Buffalo <vsbuffaloAAAAAA@gmail.com>
|  | | Date:   Sat Sep 28 23:09:01 2013 -0700
|  | |
|  | |     added ref genome download date and link
|  | |
|  | * | commit 08ccd3b056785513442fc405f568e61306d62d4b
|  | /| Author: Vince Buffalo <vsbuffaloAAAAAA@gmail.com>
|  |  | Date:   Sat Sep 28 23:10:39 2013 -0700
|  |  |
|  |  |     added reference download date
|  |  |

```

Merge conflicts are intimidating at first, but following the four-step strategy introduced at the beginning of this section will get you through it. Remember to repeatedly check `git status` to see what needs to be resolved, and use `git add` to stage edited files as you resolve the conflicts in them. At any point, if you’re overwhelmed, you can abort a merge with `git merge --abort` and start over (but beware: you’ll lose any changes you’ve made).

There’s one important caveat to merging: if your project’s code is spread out across a few files, resolving a merge conflict does not guarantee that your code *works*. Even if Git can fast-forward your local repository after a pull, it’s still possible your collaborator’s changes may break something (such is the danger when working with collaborators!). It’s always wise to do some sanity checks after pulling in code.

For complex merge conflicts, you may want to use a merge tool. Merge tools help visualize merge conflicts by placing both versions side by side, and pointing out what’s

different (rather than using Git's inline notation that uses inequality and equal signs). Some commonly used merge tools include **Meld** and **Kdiff**.

More GitHub Workflows: Forking and Pull Requests

While the shared central repository workflow is the easiest way to get started collaborating with Git, GitHub suggests a slightly different workflow based on *forking* repositories. When you visit a repository owned by another user on GitHub, you'll see a “fork” link. Forking is an entirely GitHub concept—it is not part of Git itself. By forking another person's GitHub repository, you're copying their repository to your own GitHub account. You can then clone your forked version and continue development in your own repository. Changes you push from your local version to your remote repository do not interfere with the main project. If you decide that you've made changes you want to share with the main repository, you can request that your commits are pulled using a `pull request` (another feature of GitHub).

This is the workflow GitHub is designed around, and it works very well with projects with many collaborators. Development primarily takes place in contributors' own repositories. A developer's contributions are only incorporated into the main project when pulled in. This is in contrast to a shared central repository workflow, where collaborators can push their commits to the main project at their will. As a result, lead developers can carefully control what commits are added to the project, which prevents the hassle of new changes breaking functionality or creating bugs.

Using Git to Make Life Easier: Working with Past Commits

So far in this chapter we've created commits in our local repository and shared these commits with our collaborators. But our commit history allows us to do much more than collaboration—we can compare different versions of files, retrieve past versions, and tag certain commits with helpful messages.

After this point, the material in this chapter becomes a bit more advanced. Readers can skip ahead to **Chapter 6** without a loss of continuity. If you do skip ahead, bookmark this section, as it contains many tricks used to get out of trouble (e.g., restoring files, stashing your working changes, finding bugs by comparing versions, and editing and undoing commits). In the final section, we'll also cover branching, which is a more advanced Git workflow—but one that can make your life easier.

Getting Files from the Past: `git checkout`

Anything in Git that's been committed is easy to recover. Suppose you accidentally overwrite your current version of `README.md` by using `>` instead of `>>`. You see this change with `git status`:

```

$ echo "Added an accidental line" > README.md
$ cat README.md
Added an accidental line
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
directory)
#
#       modified:   README.md
#
no changes added to commit (use "git add" and/or "git commit -a")

```

This mishap accidentally wiped out the previous contents of *README.md*! However, we can restore this file by checking out the version in our last commit (the commit HEAD points to) with the command `git checkout -- <file>`. Note that you don't need to remember this command, as it's included in `git status` messages. Let's restore *README.md*:

```

$ git checkout -- README.md
$ cat README.md
Zea Mays SNP Calling Project
Project started 2013-01-03
Samples expected from sequencing core 2013-01-10

```

We downloaded the B72 reference genome (refgen3) on 2013-01-04 from <http://maizegdb.org> into ``/share/data/refgen3/``.

But beware: restoring a file this way erases all changes made to that file since the last commit! If you're curious, the cryptic `--` indicates to Git that you're checking out a file, not a branch (`git checkout` is also used to check out branches; commands with multiple uses are common in Git).

By default, `git checkout` restores the file version from HEAD. However, `git checkout` can restore any arbitrary version from commit history. For example, suppose we want to restore the version of *README.md* one commit before HEAD. The past three commits from our history looks like this (using some options to make `git log` more concise):

```

$ git log --pretty=oneline --abbrev-commit -n 3
20041ab resolved merge conflict in README.md
08ccd3b added reference download date
dafce75 added ref genome download date and link

```

Thus, we want to restore *README.md* to the version from commit 08ccd3b. These SHA-1 IDs (even the abbreviated one shown here) function as *absolute* references to your commits (similar to absolute paths in Unix like `/some/dir/path/file.txt`). We can

always refer to a specific commit by its SHA-1 ID. So, to restore *README.md* to the version from commit 08ccd3b, we use:

```
$ git checkout 08ccd3b -- README.md
$ cat README.md
Zea Mays SNP Calling Project
Project started 2013-01-03
Samples expected from sequencing core 2013-01-10
```

We downloaded refgen3 on 2013-01-04.

If we restore to get the most recent commit's version, we could use:

```
$ git checkout 20041ab -- README.md
$ git status
# On branch master
nothing to commit, working directory clean
```

Note that after checking out the latest version of the *README.md* file from commit 20041ab, nothing has effectively changed in our working directory; you can verify this using `git status`.

Stashing Your Changes: git stash

One very useful Git subcommand is `git stash`, which saves any working changes you've made since the last commit and restores your repository to the version at HEAD. You can then reapply these saved changes later. `git stash` is handy when we want to save our messy, partial progress before operations that are best performed with a clean working directory—for example, `git pull` or branching (more on branching later).

Let's practice using `git stash` by first adding a line to *README.md*:

```
$ echo "\\nAdapter file: adapters.fa" >> README.md
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
directory)
#
#       modified:   README.md
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Then, let's stash this change using `git stash`:

```
$ git stash
Saved working directory and index state WIP on master: 20041ab
resolved merge conflict in README.md
HEAD is now at 20041ab resolved merge conflict in README.md
```

Stashing our working changes sets our directory to the same state it was in at the last commit; now our project directory is clean.

To reapply the changes we stashed, use `git stash pop`:

```
$ git stash pop
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#   directory)
#
#       modified:   README.md
#
no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (785dad46104116610d5840b317f05465a5f07c8b)
```

Note that the changes stored with `git stash` are not committed; `git stash` is a separate way to store changes outside of your commit history. If you start using `git stash` a lot in your work, check out other useful stash subcommands like `git stash apply` and `git stash list`.

More git diff: Comparing Commits and Files

Earlier, we used `git diff` to look at the difference between our working directory and our staging area. But `git diff` has many other uses and features; in this section, we'll look at how we can use `git diff` to compare our current working tree to other commits.

One use for `git diff` is to compare the difference between two arbitrary commits. For example, if we wanted to compare what we have now (at HEAD) to commit `dafce75`:

```
$ git diff dafce75
diff --git a/README.md b/README.md
index 016ed0c..9491359 100644
--- a/README.md
+++ b/README.md
@@ -3,5 +3,7 @@ Project started 2013-01-03
   Samples expected from sequencing core 2013-01-10

-Maize reference genome version: refgen3, downloaded 2013-01-04 from
+We downloaded the B72 reference genome (refgen3) on 2013-01-04 from
+http://maizegdb.org into `/share/data/refgen3/`.
+
+Adapter file: adapters.fa
```

Specifying Revisions Relative to HEAD

Like writing out absolute paths in Unix, referring to commits by their full SHA-1 ID is tedious. While we can reduce typing by using the abbreviated commits (the first seven characters of the full SHA-1 ID), there's an easier way: relative ancestry references. Similar to using relative paths in Unix like `./` and `../`, Git allows you to specify commits relative to HEAD (or any other commit, with SHA-1 IDs).

The caret notation (^) represents the *parent* commit of a commit. For example, to refer to the parent of the most recent commit on the current branch (HEAD), we'd use HEAD^ (commit 08ccd3b in our examples).

Similarly, if we'd wanted to refer to our parent's parent commit (dafce75 in our example), we use HEAD^^. Our example repository doesn't have enough commits to refer to the parent of *this* commit, but if it did, we could use HEAD^^^. At a certain point, using this notation is no easier than copying and pasting a SHA-1, so a succinct alternative syntax exists: `git HEAD~<n>`, where <n> is the number of commits back in the ancestry of HEAD (including the last one). Using this notation, HEAD^^ is the same as HEAD~2.

Specifying revisions becomes more complicated with merge commits, as these have *two* parents. Git has an elaborate language to specify these commits. For a full specification, enter `git rev-parse --help` and see the “Specifying Revisions” section of this manual page.

Using `git diff`, we can also view all changes made to a file between two commits. To do this, specify both commits and the file's path as arguments (e.g., `git diff <commit> <commit> <path>`). For example, to compare our version of *README.md* across commits 269aa09 and 46f0781, we could use either:

```
$ git diff 46f0781 269aa09 README.md
# or
$ git diff HEAD~3 HEAD~2 README.md
```

This second command utilizes the relative ancestry references explained in [“Specifying Revisions Relative to HEAD” on page 101](#).

How does this help? Git's ability to compare the changes between two commits allows you to find where and how a bug entered your code. For example, if a modified script produces different results from an earlier version, you can use `git diff` to see exactly which lines differ across versions. Git also has a tool called `git bisect` to help developers find where exactly bugs entered their commit history. `git bisect` is out of the scope of this chapter, but there are some good examples in `git bisect --help`.

Undoing and Editing Commits: `git commit --amend`

At some point, you're bound to accidentally commit changes you didn't mean to or make an embarrassing typo in a commit message. For example, suppose we were to make a mistake in a commit message:

```
$ git commit -a -m "added adpters file to readme"
[master f4993e3] added adpters file to readme
1 file changed, 2 insertions(+)
```

We could easily amend our commit with:

```
$ git commit --amend
```

`git commit --amend` opens up your last commit message in your default text editor, allowing you to edit it. Amending commits isn't limited to just changing the commit message though. You can make changes to your file, stage them, and then amend these staged changes with `git commit --amend`. In general, unless you've made a mistake, it's best to just use separate commits.

It's also possible to undo commits using either `git revert` or the more advanced `git reset` (which if used improperly can lead to data loss). These are more advanced topics that we don't have space to cover in this chapter, but I've listed some resources on this issue in this chapter's *README* file on GitHub.

Working with Branches

Our final topic is probably Git's greatest feature: branching. If you're feeling overwhelmed so far by Git (I certainly did when I first learned it), you can move forward to [Chapter 6](#) and work through this section later.

Branching is much easier with Git than in other version control systems—in fact, I switched to Git after growing frustrated with another version control system's branching model. Git's branches are virtual, meaning that branching doesn't require actually copying files in your repository. You can create, merge, and share branches effortlessly with Git. Here are some examples of how branches can help you in your bioinformatics work:

- Branches allow you to experiment in your project without the risk of adversely affecting the main branch, *master*. For example, if in the middle of a variant calling project you want to experiment with a new pipeline, you can create a new branch and implement the changes there. If the new variant caller doesn't work out, you can easily switch back to the *master* branch—it will be unaffected by your experiment.
- If you're developing software, branches allow you to develop new features or bug fixes without affecting the working production version, the *master* branch. Once

the feature or bug fix is complete, you can merge it into the *master* branch, incorporating the change into your production version.

- Similarly, branches simplify working collaboratively on repositories. Each collaborator can work on their own separate branches, which prevents disrupting the *master* branch for other collaborators. When a collaborator's changes are ready to be shared, they can be merged into the *master* branch.

Creating and Working with Branches: `git branch` and `git checkout`

As a simple example, let's create a new branch called *readme-changes*. Suppose we want to make some edits to *README.md*, but we're not sure these changes are ready to be incorporated into the main branch, *master*.

To create a Git branch, we use `git branch <branchname>`. When called without any arguments, `git branch` lists all branches. Let's create the *readme-changes* branch and check that it exists:

```
$ git branch readme-changes
$ git branch
* master
  readme-changes
```

The asterisk next to `master` is there to indicate that this is the branch we're currently on. To switch to the *readme-changes* branch, use `git checkout readme-changes`:

```
$ git checkout readme-changes
Switched to branch 'readme-changes'
$ git branch
  master
* readme-changes
```

Notice now that the asterisk is next to *readme-changes*, indicating this is our current branch. Now, suppose we edit our *README.md* section extensively, like so:

```
# Zea Mays SNP Calling Project
Project started 2013-01-03.

## Samples
Samples downloaded 2013-01-11 into `data/seqs`:

    data/seqs/zmaysA_R1.fastq
    data/seqs/zmaysA_R2.fastq
    data/seqs/zmaysB_R1.fastq
    data/seqs/zmaysB_R2.fastq
    data/seqs/zmaysC_R1.fastq
    data/seqs/zmaysC_R2.fastq

## Reference
```

We downloaded the B72 reference genome (refgen3) on 2013-01-04 from <http://maizegdb.org> into `/share/data/refgen3/``.

Now if we commit these changes, our commit is added to the *readme-changes* branch. We can verify this by switching back to the *master* branch and seeing that this commit doesn't exist:

```
$ git commit -a -m "reformatted readme, added sample info" ❶
[readme-changes 6e680b6] reformatted readme, added sample info
1 file changed, 12 insertions(+), 3 deletions(-)
$ git log --abbrev-commit --pretty=oneline -n 3 ❷
6e680b6 reformatted readme, added sample info
20041ab resolved merge conflict in README.md
08ccd3b added reference download date
$ git checkout master ❸
Switched to branch 'master'
$ git log --abbrev-commit --pretty=oneline -n 3 ❹
20041ab resolved merge conflict in README.md
08ccd3b added reference download date
dafce75 added ref genome download date and link
```

- ❶ Our commit, made on the branch *readme-changes*.
- ❷ The commit we just made (6e680b6).
- ❸ Switching back to our *master* branch.
- ❹ Our last commit on *master* is 20041ab. Our changes to *README.md* are only on the *readme-changes* branch, and when we switch back to master, Git swaps our files out to those versions on *that* branch.

Back on the *master* branch, suppose we add the *adapters.fa* file, and commit this change:

```
$ git branch
* master
  readme-changes
$ echo ">adapter-1\\nGATGATCATTCAGCGACTACGATCG" >> adapters.fa
$ git add adapters.fa
$ git commit -a -m "added adapters file"
[master dd57e33] added adapters file
1 file changed, 2 insertions(+)
create mode 100644 adapters.fa
```

Now, both branches have new commits. This situation looks like [Figure 5-6](#).

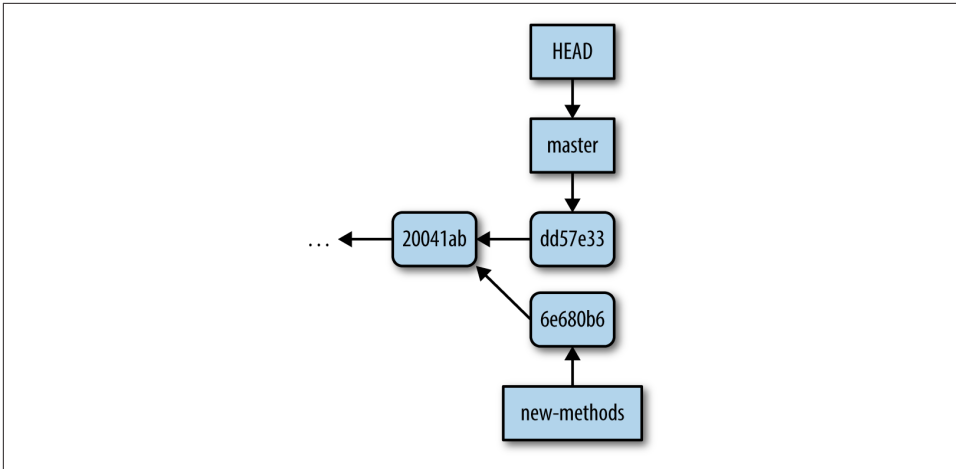


Figure 5-6. Our two branches (within Git, branches are represented as pointers at commits, as depicted here), the *master* and *readme-changes* branches have diverged, as they point to different commits (our *HEAD* points to *master*, indicating this is the current branch we’re on)

Another way to visualize this is with `git log`. We’ll use the `--branches` option to specify we want to see all branches, and `-n 2` to only see these last commits:

```
$ git log --abbrev-commit --pretty=oneline --graph --branches -n2
* dd57e33 added adapters file
| * 6e680b6 reformatted readme, added sample info
|/
```

Merging Branches: `git merge`

With our two branches diverged, we now want to merge them together. The strategy to merge two branches is simple. First, use `git checkout` to switch to the branch we want to merge the other branch into. Then, use `git merge <otherbranch>` to merge the other branch into the current branch. In our example, we want to merge the *readme-changes* branch into *master*, so we switch to *master* first. Then we use:

```
$ git merge readme-changes
Merge made by the 'recursive' strategy.
 README.md | 15 ++++++-----
 1 file changed, 12 insertions(+), 3 deletions(-)
```

There wasn’t a merge conflict, so `git merge` opens our text editor and has us write a merge commit message. Once again, let’s use `git log` to see this:

```
$ git log --abbrev-commit --pretty=oneline --graph --branches -n 3
* e9a81b9 Merge branch 'readme-changes'
| \
| * 6e680b6 reformatted readme, added sample info
```

```
* | dd57e33 added adapters file
|/
```

Bear in mind that merge conflicts can occur when merging branches. In fact, the merge conflict we encountered in “[Merge Conflicts](#)” on page 92 when pulling in remote changes was a conflict between two branches; `git pull` does a merge between a remote branch and a local branch (more on this in “[Branches and Remotes](#)” on page 106). Had we encountered a merge conflict when running `git merge`, we’d follow the same strategy as in “[Merge Conflicts](#)” on page 92 to resolve it.

When we’ve used `git log` to look at our history, we’ve only been looking a few commits back—let’s look at the entire Git history now:

```
$ git log --abbrev-commit --pretty=oneline --graph --branches
*   e9a81b9 Merge branch 'readme-changes'
| \
| * 6e680b6 reformatted readme, added sample info
* | dd57e33 added adapters file
|/
*   20041ab resolved merge conflict in README.md
| \
| * dafce75 added ref genome download date and link
* | 08ccd3b added reference download date
|/
* 269aa09 added reference genome info
* 46f0781 added information about samples
* c509f63 added .gitignore
* e4feb22 added markdown extensions to README files
* 94e2365 added information about project to README
* 3d7ffa6 initial import
```

Note that we have two bubbles in our history: one from the merge conflict we resolved after `git pull`, and the other from our recent merge of the *readme-changes* branch.

Branches and Remotes

The branch we created in the previous section was entirely local—so far, our collaborators are unable to see this branch or its commits. This is a nice feature of Git: you can create and work with branches to fit your workflow needs without having to share these branches with collaborators. In some cases, we do want to share our local branches with collaborators. In this section, we’ll see how Git’s branches and remote repositories are related, and how we can work on local branches with collaborators.

Remote branches are a special type of local branch. In fact, you’ve already interacted with these remote branches when you’ve pushed to and pulled from remote repositories. Using `git branch` with the option `--all`, we can see these hidden remote branches:

```
$ git branch --all
* master
  readme-changes
  remotes/origin/master
```

remotes/origin/master is a remote branch—we can't do work on it, but it can be synchronized with the latest commits from the remote repository using `git fetch`. Interestingly, a `git pull` is nothing more than a `git fetch` followed by a `git merge`. Though a bit technical, understanding this idea will greatly help you in working with remote repositories and remote branches. Let's step through an example.

Suppose that Barbara is starting a new document that will detail all the bioinformatics methods of your project. She creates a *new-methods* branch, makes some commits, and then pushes these commits on this branch to our central repository:

```
$ git push origin new-methods
Counting objects: 4, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 307 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@github.com:vsbuffalo/zmays-snps.git
 * [new branch]      new-methods -> new-methods
```

Back in our repository, we can fetch Barbara's latest branches using `git fetch <remote name>`. This creates a new remote branch, which we can see with `git branch --all`:

```
$ git fetch origin
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 1), reused 3 (delta 1)
Unpacking objects: 100% (3/3), done.
From github.com:vsbuffalo/zmays-snps
 = [up to date]      master      -> origin/master
 * [new branch]      new-methods -> origin/new-methods
$ git branch --all
* master
  new-methods
  remotes/origin/master
  remotes/origin/new-methods
```

`git fetch` doesn't change any of your local branches; rather, it just synchronizes your remote branches with the newest commits from the remote repositories. If after a `git fetch` we wanted to incorporate the new commits on our remote branch into our local branch, we would use a `git merge`. For example, we could merge Barbara's *new-methods* branch into our *master* branch with `git merge origin/new-methods`, which emulates a `git pull`.

However, Barbara's branch is just getting started—suppose we want to develop on the *new-methods* branch before merging it into our *master* branch. We cannot develop on remote branches (e.g., our *remotes/origin/new-methods*), so we need to make a new branch that *starts* from this branch:

```
$ git checkout -b new-methods origin/new-methods
Branch new-methods set up to track remote branch new-methods from origin.
Switched to a new branch 'new-methods'
```

Here, we've used `git checkout` to simultaneously create and switch a new branch using the `-b` option. Note that Git notified us that it's *tracking* this branch. This means that this local branch knows which remote branch to push to and pull from if we were to just use `git push` or `git pull` without arguments. If we were to commit a change on this branch, we could then push it to the remote with `git push`:

```
$ echo "\\n(1) trim adapters\\n(2) quality-based trimming" >> methods.md
$ git commit -am "added quality trimming steps"
[new-methods 5f78374] added quality trimming steps
 1 file changed, 3 insertions(+)
$ git push
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 339 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@github.com:vsbuffalo/zmays-snps.git
6364ebb..9468e38 new-methods -> new-methods
```

Development can continue on *new-methods* until you and your collaborator decide to merge these changes into *master*. At this point, this branch's work has been incorporated into the main part of the project. If you like, you can delete the remote branch with `git push origin :new-methods` and your local branch with `git branch -d new-methods`.

Continuing Your Git Education

Git is a massively powerful version control system. This chapter has introduced the basics of version control and collaborating through pushing and pulling, which is enough to apply to your daily bioinformatics work. We've also covered some basic tools and techniques that can get you out of trouble or make working with Git easier, such as `git checkout` to restore files, `git stash` to stash your working changes, and `git branch` to work with branches. After you've mastered all of these concepts, you may want to move on to more advanced Git topics such as rebasing (`git rebase`), searching revisions (`git grep`), and submodules. However, none of these topics are required in daily Git use; you can search out and learn these topics as you need them. A great resource for these advanced topics is Scott Chacon and Ben Straub's *Pro Git* book.

O'REILLY®



Early Release

RAW & UNEDITED

Python Data Science Handbook

TOOLS AND TECHNIQUES FOR DEVELOPERS

Jake VanderPlas

Python Data Science Handbook

Jake VanderPlas

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Introduction to NumPy

The next two chapters outline techniques for effectively working with data in Python. The topic is very broad: datasets can come from a wide range of sources and a wide range of formats: they could be collections of documents, collections of images, collections of sound clips, collections of numerical measurements, or nearly anything else. Despite this apparent heterogeneity, it will help us to think of all data fundamentally as arrays of numbers.

For example, images, and in particular digital images, can be thought of as simply a two dimensional array of numbers representing pixel brightness across the area. Sound clips can be thought of as one-dimensional arrays of intensity versus time. Text can be converted in various ways into numerical representations, perhaps binary digits representing the frequency of certain words or pairs of words. No matter what the data is, the first step in making it analyzable will be to transform it into arrays of numbers.

For this reason, efficient storage and manipulation of numerical arrays is absolutely fundamental to the process of doing data science. Here we'll take a look at the specialized tools that Python has to handle such numerical arrays: the *NumPy* package, and the *Pandas* package.

This chapter will cover NumPy in detail. NumPy, short for “Numerical Python”, provides an efficient interface to store and operate on dense data buffers. In many ways, NumPy arrays are like Python's built-in `list` type, but NumPy arrays provide much more efficient storage and data operations as the size of the arrays grow larger. NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python, so time spent learning to use NumPy effectively will be valuable no matter what aspect of data science interests you.

If you followed the advice from the preface and installed the Anaconda stack, you have NumPy installed and ready to go. If you're more the do-it-yourself type, you can navigate to <http://www.numpy.org/> and follow the installation instructions found there. Once you do, you can import numpy and double-check the version:

```
import numpy
numpy.__version__
'1.9.1'
```

For the pieces of the package discussed here, I'd recommend NumPy version 1.8 or later. By convention, you'll find that most people in the SciPy/PyData world will import numpy using np as an alias:

```
import numpy as np
```

Throughout this chapter, and indeed the rest of the book, you'll find that this is the way we will import and use NumPy.

Reminder about Built-in Documentation

As you read through this chapter, don't forget that IPython gives you the ability to quickly explore the contents of a package (by using the tab-completion feature), as well as the documentation of various functions (using the "?" character).

For example, you can type

```
In [3]: np.<TAB>
```

to display all the contents of the numpy namespace, and

```
In [4]: np?
```

to display NumPy's built-in documentation. More detailed documentation, along with tutorials and other resources, can be found at <http://www.numpy.org>.

Understanding Data Types in Python

Effective data-driven science and computation requires understanding how data is stored and manipulated. This section outlines and contrasts how arrays of data are handled in the Python language itself, and how NumPy improves on this. Understanding this difference is fundamental to understanding much of the material throughout the rest of the book.

Users of Python are often drawn-in by its ease of use, one piece of which is dynamic typing. While a statically-typed language like C or Java requires each variable to be explicitly declared, a dynamically-typed language like Python skips this specification.

For example, in C you might specify a particular operation as follows:

```

/* C code */
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}

```

While in Python the equivalent operation could be written this way:

```

# Python code
result = 0
for i in range(100):
    result += i

```

Notice the main difference: in C, the data types of each variable are explicitly declared, while in Python the types are dynamically inferred. This difference certainly contributes to the ease Python provides for translating algorithms to code, but to really understand data in Python we must first understand what is happening under the hood.

As we have mentioned several times, Python variables are dynamically typed. This means, for example, that we can assign any kind of data to any variable:

```

# Python code
x = 4
x = "four"

```

Here we've switched the contents of `x` from an integer to a string.

The same thing in C would lead (depending on compiler settings) to a compilation error or other unintended consequences:

```

/* C code */
int x = 4;
x = "four"; // FAILS

```

This sort of flexibility is one piece that makes Python and other dynamically-typed languages convenient and easy to use. Understanding *how* this works is an important piece of learning to analyze data efficiently and effectively with Python. But what this type-flexibility also points to is the fact that Python variables are more than just their value; they also contain extra information about the type of the value. We'll explore this more below.

A Python Integer is More than just an Integer

The standard Python implementation is written in C. This means that every Python object is simply a cleverly-disguised C structure, which contains not only its value, but other information as well.

For example, when we define an integer in Python,

```

x = 10000

```

`x` is not just a “raw” integer. It’s actually a pointer to a compound C structure, which contains several values. Looking through the Python 3.4 source code, we find that the long integer type definition effectively looks like this (once the C macros are expanded):

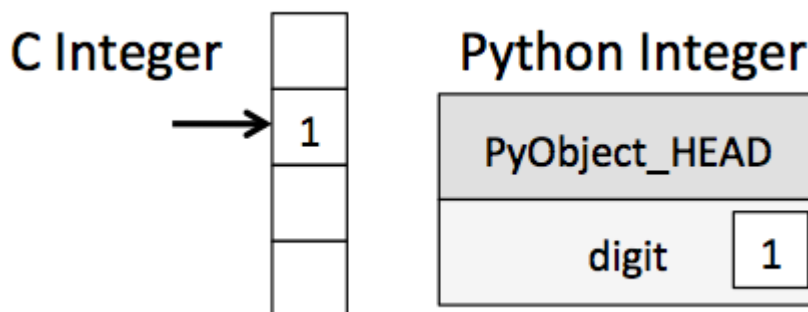
```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

A single integer in Python 3.4 actually contains four pieces:

- `ob_refcnt`, a reference count which helps Python silently handle memory allocation and deallocation
- `ob_type`, which encodes the type of the variable
- `ob_size`, which specifies the size of the following data members
- `ob_digit`, which contains the actual integer value that we expect the Python variable to represent.

This means that there is some overhead in storing an integer in Python as compared to an integer in a compiled language like C.

We can visualize this as follows:



Here `PyObject_HEAD` is the part of the structure containing the reference count, type code, and other pieces mentioned above.

Notice the difference here: a C integer is essentially a label for a position in memory whose bytes encode an integer value. A Python integer is a pointer to a position in memory containing all the Python object information, including the bytes which contain the integer value. This extra information in the Python integer structure is what allows Python to be coded so freely and dynamically. All this additional infor-

mation in Python types comes at a cost, however, which becomes especially apparent in structures which combine many of these objects.

A Python List is More than just a List

Let's consider now what happens when we use a Python data structure which holds many Python objects. The standard mutable multi-element container in Python is the list. We can create a list of integers as follows:

```
L = list(range(10))
L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
type(L[0])
int
```

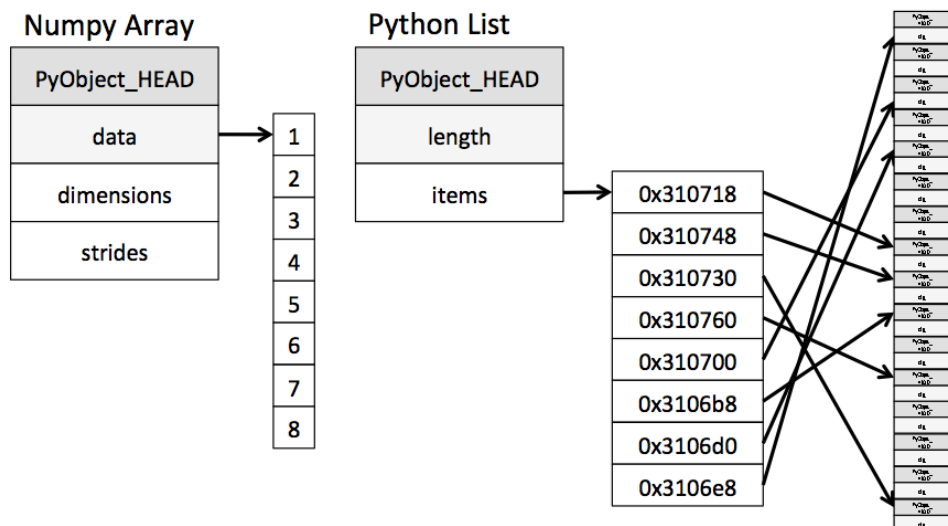
Or similarly a list of strings:

```
L2 = [str(c) for c in L]
L2
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
type(L2[0])
str
```

Because of Python's dynamic typing, we can even create heterogeneous lists:

```
L3 = [True, "2", 3.0, 4]
[type(item) for item in L3]
[bool, str, float, int]
```

But this flexibility comes at a cost: to allow these flexible types, each item in the list must contain its own type info, reference count, and other information; that is, each item is a complete Python object. In the special case that all variables are of the same type, much of this information is redundant: it can be much more efficient to store data in a fixed-type array. The difference between a dynamic-type list and a fixed-type (NumPy-style) array is illustrated in the following figure:



At the implementation level, the array essentially contains a single pointer to one contiguous block of data. The Python list, on the other hand, contains a pointer to a block of pointers, each of which in turn points to a full Python object like the Python integer we saw above. Again, the advantage of the list is flexibility: because each list element is a full structure containing both data and type information, the list can be filled with data of any desired type. Fixed-type NumPy-style arrays lack this flexibility, but are much more efficient for storing and manipulating data.

Fixed-type arrays in Python

Python offers several different options for storing data in efficient, fixed-type data buffers. Built-in since Python 3.3 is the `array` module, which can be used to create dense arrays of a uniform type:

```
import array
L = list(range(10))
A = array.array('i', L)
A
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Here `'i'` is a type code indicating the contents are integers. Much more useful, however, is the `ndarray` object of the NumPy package. While Python's `array` object provides efficient storage of array-based data, NumPy adds to this efficient *operations* on that data. We will explore these operations in later sections; here we'll demonstrate several ways of creating a numpy array.

We'll start with the standard NumPy import, under the alias `np`:

```
import numpy as np
```

Creating Arrays from Python Lists

First, we can use `np.array` to create arrays from Python lists:

```
# integer array:
np.array([1, 4, 2, 5, 3])
array([1, 4, 2, 5, 3])
```

Remember that unlike Python lists, NumPy is constrained to arrays which all contain the same type. If types do not match, NumPy will upcast if possible (here, integers are up-cast to floating point):

```
np.array([3.14, 4, 2, 3])
array([ 3.14,  4. ,  2. ,  3. ])
```

If we want to explicitly set the datatype of the resulting array, we can use the `dtype` keyword:

```
np.array([1, 2, 3, 4], dtype='float32')
array([ 1.,  2.,  3.,  4.], dtype=float32)
```

Finally, unlike Python lists, NumPy arrays can explicitly be multi-dimensional; here's one way of initializing a multidimensional array using a list of lists:

```
# nested lists result in multi-dimensional arrays
np.array([range(i, i + 3) for i in [2, 4, 6]])
array([[2, 3, 4],
       [4, 5, 6],
       [6, 7, 8]])
```

The inner lists are treated as rows of the resulting two-dimensional array.

Creating arrays from scratch

Especially for larger arrays, it is more efficient to create arrays from scratch using routines built-in to NumPy. Here are several examples:

```
# Create a length-10 integer array filled with zeros
np.zeros(10, dtype=np.int)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

# Create a (3 x 5) floating-point array filled with ones
np.ones((3, 5), dtype=float)
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])

# Create a (3 x 5) array filled with 3.14
np.full((3, 5), 3.14)
array([[ 3.14,  3.14,  3.14,  3.14,  3.14],
       [ 3.14,  3.14,  3.14,  3.14,  3.14],
       [ 3.14,  3.14,  3.14,  3.14,  3.14]])
```



```

# Create an array filled with a linear sequence
# Starting at 0, ending at 20, stepping by 2
# (this is similar to the built-in range() function)
np.arange(0, 20, 2)
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])

# Create an array of 5 values evenly spaced between 0 and 1
np.linspace(0, 1, 5)
array([ 0. ,  0.25,  0.5 ,  0.75,  1.  ])

# Create a (3 x 3) array of uniformly-distributed
# random values between 0 and 1
np.random.random((3, 3))
array([[ 0.57553592,  0.91443811,  0.50999268],
       [ 0.59732101,  0.10454524,  0.23510437],
       [ 0.10431603,  0.0562055 ,  0.47216143]])

# Create a (3 x 3) array of normally-distributed random values
# with mean 0 and standard deviation 1
np.random.normal(0, 1, (3, 3))
array([[ 0.16103055, -0.97851038, -4.40762392],
       [-0.55868781,  0.87953933,  1.95440309],
       [ 0.19103472,  0.61948762,  0.19737634]])

# Create a (3 x 3) array of random integers in the interval
# [0, 10)
np.random.randint(0, 10, (3, 3))
array([[1, 3, 0],
       [6, 0, 3],
       [5, 9, 7]])

# Create a (3 x 3) identity matrix
np.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])

# Create an uninitialized array of 3 integers
# The values will be whatever happens to already exist at that memory location
np.empty(3)
array([ 1.,  1.,  1.])

```

NumPy Standard Data Types

Because NumPy arrays contain values of a single type, it is important to have detailed knowledge of those types and their limitations. Because NumPy is built in C, the types will be familiar to users of C, Fortran, and other related languages.

The standard numpy data types are listed in the following table. Note that when constructing an array, they can be specified using a string, e.g.

```
np.zeros(10, dtype='int16')
```

or using the associated numpy object, e.g.

```
np.zeros(10, dtype=np.int16)
```

Data type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for complex128.
complex64	Complex number, represented by two 32-bit floats
complex128	Complex number, represented by two 64-bit floats

More advanced type specification is possible, such as specifying big or little endian numbers; for more information please refer to the numpy documentation at <http://numpy.org/>. NumPy also supports compound data types, which will be covered in section X.X.

The Basics of NumPy Arrays

Data manipulation in Python is nearly synonymous with NumPy array manipulation: even newer tools like Pandas (Chapter X.X) are built around the NumPy array. This section will give several examples of manipulation of NumPy arrays to access data and subarrays, and to split, reshape, and join the arrays. While the types of operations shown here may seem a bit dry and pedantic, they comprise the building-blocks of many other examples used throughout the book. Get to know them well!

We'll cover a few categories of basic array manipulations here:

- **Attributes of arrays:** determining the size, shape, memory consumption, and data types of arrays
- **Indexing of arrays:** getting and setting the value of individual array elements
- **Slicing of arrays:** getting and setting smaller subarrays within a larger array
- **Reshaping of arrays:** changing the shape of a given array
- **Joining and splitting of arrays:** combining multiple arrays into one, and splitting one array into many

NumPy Array Attributes

First let's go over some useful array attributes. We'll start by defining three random arrays, a 1-dimensional, 2-dimensional, and 3-dimensional array. We'll use NumPy's random number generator, which we will *seed* with a set value: this will ensure that the same random arrays are generated each time this code is run.

```
import numpy as np
np.random.seed(0) # seed for reproducibility

x1 = np.random.randint(10, size=6) # 1D array
x2 = np.random.randint(10, size=(3, 4)) # 2D array
x3 = np.random.randint(10, size=(3, 4, 5)) # 3D array
```

Each array has attributes `ndim` (giving the number of dimensions), `shape` (giving the size of each dimension), and `size` (giving the total size of the array):

```
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

Another useful attribute is the `dtype`, short for the data type of the array, which we discussed in the previous section:

```
print("dtype:", x3.dtype)
dtype: int64
```

(See the table of built-in data types in Section X.X).

Other attributes include `itemsize`, which lists the size (in bytes) of each array element, and `nbytes`, which lists the total size (in bytes) of the array:

```
print("itemsize:", x3.itemsize, "bytes")
print("nbytes:", x3.nbytes, "bytes")
itemsize: 8 bytes
nbytes: 480 bytes
```

In general, we expect that `nbytes` is equal to `itemsize` times `size`.

Array Indexing: Accessing Single Elements

We saw previously that individual items in Python lists can be accessed with square brackets and a zero-based integer index; NumPy arrays use similar notation with some additional enhancements for arrays with multiple dimensions.

In a one-dimensional array, the i^{th} can be accessed by specifying the desired index in square brackets, just as with Python lists:

```
x1
array([5, 0, 3, 3, 7, 9])

x1[0]
5

x1[4]
7
```

To index from the end of the array, you can use negative indices:

```
x1[-1]
9

x1[-2]
7
```

In a multi-dimensional array, items can be accessed using a comma-separated tuple of indices:

```
x2
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])

x2[0, 0]
3

x2[2, 0]
1

x2[2, -1]
7
```

Values can also be modified using any of the above index notation:

```
x2[0, 0] = 12
x2
array([[12, 5, 2, 4],
       [ 7, 6, 8, 8],
       [ 1, 6, 7, 7]])
```

Unlike Python lists, though, keep in mind that NumPy arrays have a fixed type! This means, for example, that if you attempt to insert a floating point value to an integer array, the value will be silently truncated. Don't be caught unaware by this behavior!

```
x1[0] = 3.14159 # this will be truncated!
x1
array([3, 0, 3, 3, 7, 9])
```

Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (:) character. The NumPy slicing syntax follows that of the standard Python list: to access a slice of an array *x*, use

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values `start=0`, `stop=(size of dimension)`, `step=1`. We'll take a look at accessing sub-arrays in one dimension and in multiple dimensions:

One-dimensional Subarrays

```
x = np.arange(10)
x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

x[:5] # first five elements
array([0, 1, 2, 3, 4])

x[5:] # elements after index 5
array([5, 6, 7, 8, 9])

x[4:7] # middle sub-array
array([4, 5, 6])

x[::2] # every other element
array([0, 2, 4, 6, 8])

x[1::2] # every other element, starting at index 1
array([1, 3, 5, 7, 9])
```

A potentially confusing case is when the `step` value is negative. In this case, the defaults for `start` and `stop` are swapped. This becomes a convenient way to reverse an array:

```
x[::-1] # all elements, reversed
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])

x[5::-2] # reversed every other from index 5
array([5, 3, 1])
```

Multi-dimensional Subarrays

Multi-dimensional slices work in the same way, except multiple slices can be specified. For example:

```

x2
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])

x2[:2, :3] # two rows, three columns
array([[12,  5,  2],
       [ 7,  6,  8]])

x2[:3, ::2] # all rows, every other column
array([[12,  2],
       [ 7,  8],
       [ 1,  7]])

```

Finally, subarray dimensions can even be reversed together:

```

x2[::-1, ::-1]
array([[ 7,  7,  6,  1],
       [ 8,  8,  6,  7],
       [ 4,  2,  5, 12]])

```

Accessing Array Rows and Columns. One commonly-needed routine is accessing of single rows or columns of an array. This can be done by combining indexing and slicing, using an empty slice marked by a single colon (:):

```

print(x2[:, 0]) # first column of x2
[12  7  1]

print(x2[0, :]) # first row of x2
[12  5  2  4]

```

In the case of row access, the empty slice can be left-out for a more compact syntax:

```

print(x2[0]) # equivalent to x2[0, :]
[12  5  2  4]

```

Subarrays as no-copy views

One important – and extremely useful – thing to know about array slices is that they return *views* rather than *copies* of the array data. (If you’re familiar with Python lists, keep in mind that this is different behavior: In lists, slices are copies by default) Consider our two-dimensional array from above:

```

print(x2)
[[12  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]

```

Let’s extract a 2×2 subarray from this:

```

x2_sub = x2[:2, :2]
print(x2_sub)
[[12  5]
 [ 7  6]]

```

Now if we modify this sub-array, we'll see that the original array is changed! Observe:

```
x2_sub[0, 0] = 99
print(x2_sub)
[[99  5]
 [ 7  6]]

print(x2)
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

This is actually an extremely useful default behavior: it means that when we work with large datasets, we can access and process pieces of these datasets without copying the underlying data buffer. This works because NumPy arrays have a very flexible internal representation, which we'll explore in more detail in section X.X.

Creating Copies of Arrays

Despite the nice features of array views, it is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the `copy()` method:

```
x2_sub_copy = x2[:2, :2].copy()
print(x2_sub_copy)
[[99  5]
 [ 7  6]]
```

If we now modify this subarray, the original array is not touched:

```
x2_sub_copy[0, 0] = 42
print(x2_sub_copy)
[[42  5]
 [ 7  6]]

print(x2)
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Reshaping of Arrays

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the `reshape` method. For example, if you want to put the numbers 1 through 9 in a 3x3 grid, you can do the following:

```
grid = np.arange(1, 10).reshape((3, 3))
print(grid)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Note that for this to work, the size of the initial array must match the size of the reshaped array. Where possible, the `reshape` method will use a no-copy view of the initial array, but with non-contiguous memory buffers this is not always the case.

Another common reshaping pattern is the conversion of a 1D array into a 2D row or column matrix. This can be done with the `reshape` method, or more easily done by making use of the `newaxis` keyword within a slice operation:

```
x = np.array([1, 2, 3])

# row vector via reshape
x.reshape((1, 3))
array([[1, 2, 3]])

# row vector via newaxis
x[np.newaxis, :]
array([[1, 2, 3]])

# column vector via reshape
x.reshape((3, 1))
array([[1],
       [2],
       [3]])

# column vector via newaxis
x[:, np.newaxis]
array([[1],
       [2],
       [3]])
```

We'll make use of these types of transformations often through the remainder of the text.

Array Concatenation and Splitting

All of the above routines worked on single arrays. It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays. We'll take a look at those operations here.

Concatenation of Arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines `np.concatenate`, `np.vstack`, and `np.hstack`. `Concatenate` takes a tuple or list of arrays as its first argument, as we can see here:

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
array([1, 2, 3, 3, 2, 1])
```

You can also concatenate more than two arrays at once:


```

z = [99, 99, 99]
print(np.concatenate([x, y, z]))
[ 1  2  3  3  2  1 99 99 99]

```

It can also be used for two-dimensional arrays:

```

grid = np.array([[1, 2, 3],
                 [4, 5, 6]])

# concatenate along the first axis
np.concatenate([grid, grid])
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])

# concatenate along the second axis (zero-indexed)
np.concatenate([grid, grid], axis=1)
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])

```

For working with arrays of mixed dimensions, it can be clearer to use the `np.vstack` (vertical stack) and `np.hstack` (horizontal stack) functions:

```

x = np.array([1, 2, 3])
grid = np.array([[9, 8, 7],
                 [6, 5, 4]])

# vertically stack the arrays
np.vstack([x, grid])
array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])

# horizontally stack the arrays
y = np.array([[99],
              [99]])
np.hstack([grid, y])
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99]])

```

Similarly, `np.dstack` will stack three-dimensional arrays along the third axis.

Splitting of Arrays

The opposite of concatenation is splitting, which is implemented by the functions `np.split`, `np.hsplit`, and `np.vsplit`. For each of these, we can pass a list of indices giving the split points:

```

x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
[1 2 3] [99 99] [3 2 1]

```

Notice that N split-points, leads to $N + 1$ subarrays. The related functions `np.hsplit` and `np.vsplit` are similar:

```
grid = np.arange(16).reshape((4, 4))
grid
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])

upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]

left, right = np.hsplit(grid, [2])
print(left)
print(right)
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

Similarly, `np.dsplit` will split arrays along the third axis.

Summary

This section has covered many of the basic patterns used in NumPy to examine the content of arrays, to access elements and portions of arrays, to reshape arrays, and to join and split arrays. These operations are the building-blocks of more sophisticated recipes that we'll see later in the book. They may seem a bit dry and pedantic, but these patterns are a bit like the conjugations and grammar rules that must be memorized when first learning a foreign language: an absolutely essential foundation for the more interesting material to come. Get to know these patterns well!

Random Number Generation

Often in computational science it is useful to be able to generate sequences of random numbers. This might seem simple, but after scratching at the surface it reveals itself to be a much more subtle problem: computers are entirely deterministic machines: can a computer algorithm ever be said to generate anything truly random? Further, how might we even define what traits a random sequence would have?

Given these questions, many purists will make fine distinctions between random numbers, pseudo-random numbers, quasi-random numbers, etc. Here we'll follow Press et al. [REF] and ignore such subtleties in favor of a more practical (if perhaps less satisfying) definition:

the deterministic program that produces a random sequence should be different from, and — in all measurable respects — statistically uncorrelated with, the computer program that uses its output.

That is, a sequence of random numbers is random if it's random enough for the purposes we're using it for. This is an admittedly circular definition, but ends up being very useful in practice. This section outlines how to generate random numbers and arrays of random numbers within Python. After exploring a simple Python implementation of a deterministic pseudorandom number generator, it covers the efficient random number tools available in Python and NumPy.

Understanding a Simple “Random” Sequence

Before exploring the tools available in Python and NumPy, we'll briefly code from-scratch a simple pseudorandom number generator. The code in this section is for example only, and should not be used in practice: below we'll see examples of the much more efficient and robust random number generators available in Python. Here the goal is to give some insight into the principles behind these algorithms and the resulting strengths and weaknesses.

While it is possible to obtain sequences of truly random numbers through physical means (by, e.g. monitoring decay events of a radioactive substance) most algorithmic random number generators used in practice are simple deterministic algorithms designed to produce a sequence of pseudo-random numbers with suitable properties. The simplest of these are *linear congruential generators*, which generate a sequence of integers r_i using the following recursive form:

$$r_{i+1} = (ar_i + c) \text{ MOD } m$$

where a , c , and m are chosen such that the resulting numbers have useful properties. For example, we can follow Knuth and Lewis [REF] who suggest generating 32-bit uniform deviates using $m = 2^{32}$, $a = 1664525$, and $c = 1013904223$. We can implement this step simply in Python:

```
def LCG_next(r, a=1664525, c=1013904223, m=2 ** 32):
    """
    Generate the next pseudorandom number
    using a Linear Congruential Generator
    """
    return (a * r + c) % m

seed = 0
for i in range(5):
```

```

    seed = LCG_next(seed)
    print(seed)
1013904223
1196435762
3519870697
2868466484
1649599747

```

Often, rather than creating a sequence of integers, it is more convenient to create a sequence of *uniform deviates*: that is, numbers distributed uniformly in the half-open interval $[0, 1)$. Here we'll implement this by making use of Python's convenient generator syntax (see section X.X). We'll also add code which will automatically seed the sequence based on the current microseconds in the system clock.

```

from datetime import datetime
from itertools import islice

def LCG_generator(seed=None, a=1664525, c=1013904223, m=2 ** 32):
    """
    Linear Congruential generator of pseudorandom numbers
    """
    if seed is None:
        # If seed is not provided, use current microseconds
        seed = datetime.now().microsecond

    while True:
        seed = LCG_next(seed, a=a, c=c, m=m)
        yield float(seed) / m

def simple_uniform_deviate(N, seed=None):
    """
    return a list of N pseudorandom numbers
    in the interval [0, 1)
    """
    gen = LCG_generator(seed)
    return list(islice(gen, N))

# Print a list of 5 random numbers
simple_uniform_deviate(5)
[0.7377541123423725,
 0.3999146604910493,
 0.18632183666341007,
 0.591240135487169,
 0.22258975286968052]

```

Again, the above code is *not* an ideal way to generate sequences of random numbers; it is included to give you a bit of insight into *how* pseudorandom numbers are generated in practice. We've started with a seed value, which defaults to some changing

value or other source of pseudo-randomness available in the operating system. From this seed, we construct an algorithm which creates a deterministic sequence of values which are sufficiently random for our purposes. All pseudorandom number generators share this determinism.

Below we'll see some more efficient and sophisticated random number generators built-in to Python and to NumPy. Despite their complexity, under the hood they are just deterministic algorithms which step from one seed to the next, albeit with more involved steps than we used above.

Built-in tools: Python's random module

Python has a built-in random module which uses the sophisticated Mersenne Twister algorithm [REF] to generate sequences of uniform pseudorandom numbers:

```
import random
[random.random() for i in range(5)]
[0.6307611680584317,
 0.4050472985131417,
 0.4218846620497734,
 0.7692476200181592,
 0.15093482641991562]
```

The seed is implicitly set from an operating system source such as the current time. Alternatively, we can set the seed explicitly from any hashable object:

```
random.seed(100)
[random.random() for i in range(5)]
[0.1456692551041303,
 0.45492700451402135,
 0.7707838056590222,
 0.705513226934028,
 0.7319589730332557]
```

Reseeding with the same value will lead to identical results, as we can confirm:

```
random.seed(100)
[random.random() for i in range(5)]
[0.1456692551041303,
 0.45492700451402135,
 0.7707838056590222,
 0.705513226934028,
 0.7319589730332557]
```

The built-in random module has many more functions and features; for more information refer to the Python documentation at <http://www.python.org/>. Rather than digging further into Python's random module, we'll instead move our focus to the random number tools included with NumPy. These are optimized to generate sequences of random numbers, and will end up being much more useful for the types of data science tasks we will tackle in this book.

Efficient Random Numbers: `numpy.random`

The `numpy.random` submodule, like Python's built-in `random` module, is based on the Mersenne Twister algorithm. Unlike the Python's built-in tools, it is optimized for generating large arrays of random numbers.

Uniform Deviates

The basic interface is the `rand` function, which generates uniform deviates:

```
import numpy as np
np.random.rand()
0.6065604039577245
```

Arrays of any size and shape can be created with this function:

```
np.random.rand(5)
array([ 0.11763091,  0.68041723,  0.25315934,  0.41112628,  0.17916994])

np.random.rand(3, 3)
array([[ 0.47070662,  0.19456102,  0.96443832],
       [ 0.13359021,  0.4849765 ,  0.69473637],
       [ 0.04753502,  0.88342211,  0.60243267]])
```

Random Seed

Though we did not set the seed above, it was implicitly set based on a system-dependent source of randomness. As above, specifying an explicit seed leads to reproducible sequences:

```
np.random.seed(2)
print(np.random.rand(5))

np.random.seed(2)
print(np.random.rand(5))
[ 0.4359949  0.02592623  0.54966248  0.43532239  0.4203678 ]
[ 0.4359949  0.02592623  0.54966248  0.43532239  0.4203678 ]
```

We'll use explicit seeds like this throughout this book: they can be very useful for making demonstrations and results reproducible.

Random Integers

Random integers can be obtained using the `np.random.randint` function:

```
# 10 random integers in the interval [0, 10)
np.random.randint(0, 10, size=10)
array([2, 1, 5, 4, 4, 5, 7, 3, 6, 4])
```

Note that here the upper bound is exclusive: that is, the value 10 will never appear in the above list. A related function is `np.random.random_integers`, which instead implements an inclusive upper bound:

```
# 10 random integers in the interval [0, 10]
np.random.random_integers(0, 10, size=10)
array([10, 3, 7, 6, 10, 1, 10, 3, 5, 8])
```

Permutations and Selections

Both of the above random integer functions return lists containing repeats. If instead you'd like a random permutation of non-repeating integers, you can use `np.random.permutation`:

```
x = np.arange(10)
np.random.permutation(x)
array([5, 9, 8, 0, 2, 1, 3, 7, 6, 4])
```

A related function is `np.random.choice`, which allows you to select N random values from any array, with or without replacement:

```
np.random.choice(x, 10, replace=False)
array([5, 3, 8, 6, 7, 0, 1, 9, 4, 2])

np.random.choice(x, 10, replace=True)
array([9, 8, 7, 1, 6, 8, 5, 9, 9, 9])
```

Neither of these functions modify the original array; if we'd like to shuffle the array in-place we can use the `np.random.shuffle` function:

```
print(x)
np.random.shuffle(x)
print(x)
[0 1 2 3 4 5 6 7 8 9]
[1 4 7 6 5 2 8 9 0 3]
```

Normally-distributed Values

While uniform deviates and collections of integers are useful, there are many other distribution functions that are useful in practice. Perhaps the best-known is the normal distribution, where the probability density function is

$$p(x|\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp \left[\frac{-(x-\mu)^2}{2\sigma^2} \right]$$

where μ is the mean, and σ is the standard deviation. Values can be drawn from a *standard normal*, which has $(\mu, \sigma) = (0, 1)$, using the `np.random.randn` function:

```
# Sequence of normally-distributed values
np.random.randn(10)
array([-0.52715869,  1.11385518, -1.75408685, -0.24978025,  0.54959138,
        0.52170749, -1.52343212,  0.08266405, -0.43774428,  1.57694963])
```

To specify different values of μ and σ , you can use the `np.random.normal` function:

```
# mean=10, stdev=2, size=(3, 3)
np.random.normal(10, 2, (3, 3))
```

```
array([[ 13.8069827 ,  10.05675847,   9.65668028],
       [ 13.7363806 ,   9.83213452,  11.04451775],
       [ 12.24302322,   8.483193  ,  10.34135175]])
```

Other Distributions

There are numerous other distribution functions available in the `np.random` submodule, more than we can cover here.

For example, you can create a sequence of poisson-distributed integers:

```
# poisson distribution for lambda=5
np.random.poisson(5, size=10)
array([ 6,  5,  9, 10,  3, 10,  7,  0,  4,  2])
```

You can create a sequence of exponentially-distributed values:

```
# exponential distribution with scale=1
np.random.exponential(1, size=10)
array([ 2.037982 ,  0.32218776,  0.15689987,  0.70402945,  0.4081106 ,
        0.1763738 ,  0.24755148,  1.81105024,  1.27420941,  0.05286104])
```

For information on further available random distributions refer to the documentation of the `np.random` submodule, and of the `scipy.stats.distributions` submodule.

Simultaneously Using Multiple Chains

Sometimes it is useful to have multiple random number sequences available concurrently; `numpy.random` provides the `RandomState` class for this purpose. Under the hood, the above functions are simply making use of a single global instance of this class. It can be instantiated and used as follows:

```
# instantiate a random number generator
# if seed is not specified, it will be seeded
# with a system-dependent source of randomness
rng = np.random.RandomState(seed=2)
```

Once this class instance is created, many of the above functions can be used as a method of the class. For example:

```
rng.rand(3, 3)
array([[ 0.4359949 ,  0.02592623,  0.54966248],
       [ 0.43532239,  0.4203678 ,  0.33033482],
       [ 0.20464863,  0.61927097,  0.29965467]])

rng.randint(0, 10, size=5)
array([4, 4, 5, 7, 3])

rng.randn(3, 3)
array([[ 2.6460672 , -0.04386375, -0.96561968],
       [ 0.87866389, -2.24587483,  1.11957525],
       [-1.054368 , -1.0088915 , -0.06752199]])
```


Random Numbers: Further Resources

This section has been a quick introduction to some of the pseudorandom number generators available in Python. For more information, refer to the following sources:

- <http://numpy.org>: the official NumPy documentation has much more information on the routines available in the `np.random` package
- <http://scipy.org>: the `scipy.stats.distributions` submodule contains implementations of many more advanced and obscure statistical distributions, including the ability to draw random samples from these distributions.
- Press et al. [REF] contains a more thorough discussion of (pseudo)random number generation, including common algorithms and their strengths and weaknesses.
- Ivezic et al. [REF] contains a Python-driven discussion of various distributions implemented in NumPy and SciPy, as well as examples of applications which depend on these tools.

Computation on NumPy Arrays: Universal Functions

Computation on numpy arrays can be very fast, or it can be very slow. The key to making it fast is to use *vectorized* operations, generally implemented through NumPy's *Universal Functions* (ufuncs for short). This section motivates the need for NumPy's ufuncs, which can be used to make repeated calculations on array elements much more efficient. It then introduces many of the most common and useful arithmetic ufuncs available in the NumPy package.

The Slowness of Loops

Python's default implementation (known as CPython) does some operations very slowly. This is in part due to its dynamic nature: the fact that types are not specified, so that sequences of operations cannot be compiled-down to efficient machine code, as they are in languages like C and Fortran. Recently there have been various attempts to address this weakness: well-known examples are the **PyPy** project, a just-in-time compiled implementation of Python; the **Cython** project, which converts Python code to compilable C code; and the **Numba** project, which converts snippets of Python code to fast LLVM bytecode. Each of these has its strengths and weaknesses, but it is safe to say that none of the three approaches has yet surpassed the reach and popularity of the standard CPython engine.

The slowness of Python generally manifests itself in situations where many small operations are being repeated: i.e. looping over arrays to operate on each element. For example, imagine we have an array of values and we'd like to compute the reciprocal of each. A straightforward approach might look like this:

```

import numpy as np
np.random.seed(0)

def compute_reciporicals(values):
    output = np.empty(len(values))
    for i in range(len(values)):
        output[i] = 1 / values[i]
    return output

values = np.random.randint(1, 10, size=5)
compute_reciporicals(values)
array([ 0.16666667,  1.          ,  0.25         ,  0.25         ,  0.125        ])

```

This implementation probably feels fairly natural to someone from, say, a C or Java background. But if we measure the execution time of this code for a large input, we see that this operation is very slow, perhaps surprisingly so! Here we'll use IPython's `%timeit` magic function, which was introduced in sections X.X and X.X:

```

big_array = np.random.randint(1, 100, size=1E6)
%timeit compute_reciporicals(big_array)
1 loops, best of 3: 292 ms per loop

```

It takes significant fraction of a second to compute these million operations and to store the result! When even cell phones have processing speeds measured in Giga-FLOPS (i.e. billions of numerical operations per second), this seems almost absurdly slow. It turns out that the bottleneck here is not the operations themselves, but the dynamic type-checking that CPython must do at each cycle of the loop. Each time the reciporical is computed, Python first examines the object's type and does a dynamic lookup of the correct function to use for that type. If we were working in compiled code instead, this type specification would be known before the code executes and the result could be computed much more efficiently.

Introducing UFuncs

For many types of operations, NumPy provides a convenient interface into just this kind of statically-typed, compiled routine. This is known as a *vectorized* operation. This can be accomplished by simply performing an operation on the array, which will then be applied to each element. The loop over each operation can then be pushed into the compiled layer of numpy, leading to much faster execution.

Compare the results of the following two:

```

print(compute_reciporicals(values))
print(1 / values)
[ 0.16666667  1.          0.25         0.25         0.125        ]
[ 0.16666667  1.          0.25         0.25         0.125        ]

```

Looking at the execution time for our big array, we see that it completes orders of magnitude faster than the Python loop:

```
%timeit (1 / big_array)
100 loops, best of 3: 8.09 ms per loop
```

This vectorized operation is known as a *ufunc* in NumPy, short for “Universal Function”. The main purpose of ufuncs is to quickly execute repeated operations by pushing the loops down into fast compiled code. They are extremely flexible: above we saw an operation between a scalar and an array; we can also operate between two arrays:

```
np.arange(5) / np.arange(1, 6)
array([ 0.         ,  0.5         ,  0.66666667,  0.75         ,  0.8         ])
```

And they are not limited to one-dimensional arrays: they can also act on multi-dimensional arrays as well:

```
x = np.arange(9).reshape((3, 3))
2 ** x
array([[ 1,  2,  4],
       [ 8, 16, 32],
       [64, 128, 256]])
```

Vectorized operations such as ufuncs are nearly always more efficient than their counterpart implemented using Python loops. Any time you see such a loop in a Python script, you should consider whether it can be replaced with a vectorized expression.

Exploring NumPy’s UFuncs

Ufuncs exist in two flavors: *unary ufuncs*, which operate on a single input, and *binary ufuncs*, which operate on two inputs. We’ll see examples of both these types of functions below.

Array Arithmetic

NumPy’s ufuncs feel very natural to use because they make use of Python’s native arithmetic operators. The standard addition, subtraction, multiplication, and division can all be used:

```
x = np.arange(4)
print("x =", x)
print("x + 5 =", x + 5)
print("x - 5 =", x - 5)
print("x * 2 =", x * 2)
print("x / 2 =", x / 2)
print("x // 2 =", x // 2)
x      = [0 1 2 3]
x + 5   = [5 6 7 8]
x - 5   = [-5 -4 -3 -2]
x * 2   = [0 2 4 6]
x / 2   = [ 0.  0.5  1.  1.5]
x // 2  = [0 0 1 1]
```

There is also a unary ufunc for negation, and a `**` operator for exponentiation, and a `%` operator for modulus:

```
print("-x      = ", -x)
print("x ** 2 = ", x ** 2)
print("x % 2  = ", x % 2)
-x      =  [ 0 -1 -2 -3]
x ** 2   =  [0 1 4 9]
x % 2    =  [0 1 0 1]
```

In addition, these can be strung together however you wish, and the standard order of operations is respected:

```
-(0.5*x + 1) ** 2
array([-1.   , -2.25, -4.   , -6.25])
```

Each of these arithmetic operations are simply convenient wrappers around specific functions built-in to NumPy; for example, the `+` operator is simply a wrapper for the `add` function:

```
np.add(x, 2)
array([2, 3, 4, 5])
```

The following is a table of the arithmetic operators implemented in NumPy:

Operator	Equivalent ufunc	Description
+	<code>np.add</code>	addition (e.g. $1 + 1 = 2$)
-	<code>np.subtract</code>	subtraction (e.g. $3 - 2 = 1$)
-	<code>np.negative</code>	unary negation (e.g. -2)
*	<code>np.multiply</code>	multiplication (e.g. $2 * 3 = 6$)
/	<code>np.divide</code>	division (e.g. $3 / 2 = 1.5$)
//	<code>np.floor_divide</code>	floor division (e.g. $3 // 2 = 1$)
**	<code>np.power</code>	exponentiation (e.g. $2 ** 3 = 8$)
%	<code>np.mod</code>	modulus/remainder (e.g. $9 \% 4 = 1$)

Here we are not covering boolean/bitwise operators: for a similar table of boolean operators and ufuncs, see section X.X.

Absolute value

Just as NumPy understands Python's built-in arithmetic operators, it also understands Python's built-in absolute value function:

```
x = np.array([-2, -1, 0, 1, 2])
abs(x)
array([2, 1, 0, 1, 2])
```

The corresponding NumPy ufunc is `np.absolute`, which is also available as `np.abs`:

```

np.absolute(x)
array([2, 1, 0, 1, 2])

np.abs(x)
array([2, 1, 0, 1, 2])

```

This ufunc can also handle complex data, in which the absolute value returns the magnitude:

```

x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
np.abs(x)
array([ 5.,  5.,  2.,  1.])

```

Trigonometric Functions

NumPy provides a large number of useful ufuncs, and some of the most useful for the data scientist are the trigonometric functions. We'll start by defining an array of angles:

```
theta = np.linspace(0, np.pi, 3)
```

Now we can compute some trigonometric functions on these values:

```

print("theta      = ", theta)
print("sin(theta) = ", np.sin(theta))
print("cos(theta) = ", np.cos(theta))
print("tan(theta) = ", np.tan(theta))
theta      = [ 0.          1.57079633  3.14159265]
sin(theta) = [ 0.00000000e+00  1.00000000e+00  1.22464680e-16]
cos(theta) = [ 1.00000000e+00  6.12323400e-17 -1.00000000e+00]
tan(theta) = [ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]

```

The values are computed to within machine precision, which is why we values which should be zero do not always hit exactly zero. Inverse trigonometric functions are also available:

```

x = [-1, 0, 1]
print("x          = ", x)
print("arcsin(x) = ", np.arcsin(x))
print("arccos(x) = ", np.arccos(x))
print("arctan(x) = ", np.arctan(x))
x          = [-1, 0, 1]
arcsin(x) = [-1.57079633  0.          1.57079633]
arccos(x) = [ 3.14159265  1.57079633  0.          ]
arctan(x) = [-0.78539816  0.          0.78539816]

```

Exponents and Logarithms

Another common type of operation available in a NumPy ufunc are the exponentials:

```

x = [1, 2, 3]
print("x          =", x)
print("e^x        =", np.exp(x))
print("2^x        =", np.exp2(x))

```

```

print("3^x =", np.power(3, x))
x      = [1, 2, 3]
e^x    = [ 2.71828183  7.3890561 20.08553692]
2^x    = [ 2.  4.  8.]
3^x    = [ 3  9 27]

```

The inverse of the exponentials, the logarithms, are also available. The basic `np.log` gives the natural logarithm; if you prefer to compute the base-2 logarithm or the base-10 logarithm, these are available as well:

```

x = [1, 2, 4, 10]
print("x      =", x)
print("ln(x)   =", np.log(x))
print("log2(x) =", np.log2(x))
print("log10(x) =", np.log10(x))
x      = [1, 2, 4, 10]
ln(x)   = [ 0.          0.69314718  1.38629436  2.30258509]
log2(x) = [ 0.          1.          2.          3.32192809]
log10(x) = [ 0.          0.30103    0.60205999  1.          ]

```

There are also some specialized versions which are useful for maintaining precision with very small input:

```

x = [0, 0.001, 0.01, 0.1]
print("exp(x) - 1 =", np.expm1(x))
print("log(1 + x) =", np.log1p(x))
exp(x) - 1 = [ 0.          0.0010005  0.01005017  0.10517092]
log(1 + x) = [ 0.          0.0009995  0.00995033  0.09531018]

```

When `x` is very small, these functions give more precise values than if the raw `np.log` or `np.exp` were to be used.

Specialized Ufuncs

NumPy has many more ufuncs available including hyperbolic trig functions, bitwise arithmetic, comparison operators, conversions from radians to degrees, rounding and remainders, and much more. A look through the `numpy` documentation can reveal many interesting functionality.

Another excellent source for more specialized and obscure ufuncs is the `scipy.special` submodule. There are far too many functions here to list them all, but we'll go over some highlights:

```

from scipy import special

# Bessel functions:
x = [0, 1, 2]
print("J0(x) =", special.j0(x))
print("J1(x) =", special.j1(x))
print("J4(x) =", special.jn(x, 4))
J0(x) = [ 1.          0.76519769  0.22389078]

```

```
J1(x) = [ 0.          0.44005059  0.57672481]
J4(x) = [-0.39714981 -0.06604333  0.36412815]
```

See also `special.in` (modified Bessel functions) and `special.kn` (modified Bessel functions of the second kind).

```
# Gamma functions (generalized factorials) & related functions
x = [1, 5, 10]
print("gamma(x)      =", special.gamma(x))
print("ln|gamma(x)| =", special.gammaln(x))
print("beta(x, 2)    =", special.beta(x, 2))
gamma(x)      = [ 1.000000000e+00  2.400000000e+01  3.628800000e+05]
ln|gamma(x)|  = [ 0.          3.17805383  12.80182748]
beta(x, 2)    = [ 0.5          0.03333333  0.00909091]
```

```
# Error function (Integral of Gaussian)
# its complement, and its inverse
x = np.array([0, 0.3, 0.7, 1.0])
print("erf(x)      =", special.erf(x))
print("erfc(x)     =", special.erfc(x))
print("erfinv(x)    =", special.erfinv(x))
erf(x)      = [ 0.          0.32862676  0.67780119  0.84270079]
erfc(x)     = [ 1.          0.67137324  0.32219881  0.15729921]
erfinv(x)   = [ 0.          0.27246271  0.73286908  inf]
```

There are many, many more ufuncs available in both `numpy` and `scipy.special`. Because the documentation of these packages is available online, a web search along the lines of “Gamma function Python” will generally find the relevant information.

Advanced Ufunc Features

Ufuncs are very flexible beasts; they implement several interesting and useful features, which we’ll quickly demonstrate below.

Specifying Output

For large calculations, it is sometimes useful to be able to specify the array where the result of the calculation will be stored. Rather than creating a temporary array, this can be used to write computation results directly to the memory location where you’d like them to be. For all ufuncs, this can be done using the `out` argument of the function:

```
x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y)
print(y)
[ 0. 10. 20. 30. 40.]
```

This can even be used with array views. For example, we can write the results of a computation to every other element of a specified array:

```

y = np.zeros(10)
np.power(2, x, out=y[::2])
print(y)
[ 1.  0.  2.  0.  4.  0.  8.  0. 16.  0.]

```

If we had instead written `y[::2] = 2 ** x`, this would have resulted in the creation of a temporary array to hold the results of `2 ** x`, followed by a second operation copying those values into the `y` array. This doesn't make much of a difference for such a small computation, but for very large arrays the memory savings from careful use of the `out` argument can be significant.

Aggregates

For binary ufuncs, there are some interesting aggregates that can be computed from Ufuncs. For example, if we'd like to *reduce* an array with a particular operation, we can use the `reduce` method of any ufunc. A reduce repeatedly applies a given operation to the elements of an array until only a single result is left.

For example, calling `reduce` on the `add` ufunc returns the sum of all elements in the array:

```

x = np.arange(1, 6)
np.add.reduce(x)
15

```

Similarly, calling `reduce` on the `multiply` ufunc results in the product of all array elements:

```

np.multiply.reduce(x)
120

```

If we'd like to store all the intermediate results of the computation, we can instead use `accumulate`:

```

np.add.accumulate(x)
array([ 1,  3,  6, 10, 15])

np.multiply.accumulate(x)
array([ 1,  2,  6, 24, 120])

```

Note that for these particular cases, there are dedicated NumPy functions to compute the results (`np.sum`, `np.prod`, `np.cumsum`, `np.cumprod`). We'll cover these and other aggregation functions in section X.X.

Outer Products

Finally, any ufunc can compute the output of all pairs of two different inputs using the `outer` method. This allows you, in one line, to do things like create a multiplication table:


```

x = np.arange(1, 13)
np.multiply.outer(x, x)
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12],
       [ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24],
       [ 3,  6,  9, 12, 15, 18, 21, 24, 27, 30, 33, 36],
       [ 4,  8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48],
       [ 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60],
       [ 6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72],
       [ 7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84],
       [ 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96],
       [ 9, 18, 27, 36, 45, 54, 63, 72, 81, 90, 99, 108],
       [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120],
       [11, 22, 33, 44, 55, 66, 77, 88, 99, 110, 121, 132],
       [12, 24, 36, 48, 60, 72, 84, 96, 108, 120, 132, 144]])

```

Other extremely useful methods of ufuncs are the `ufunc.at` and `ufunc.reduceat` methods, which we'll explore when we cover *fancy indexing* in section X.X.

Finding More

More information on universal functions (including the full list of available functions) can be found on the NumPy and SciPy documentation websites:

- NumPy: <http://www.numpy.org>
- SciPy: <http://www.scipy.org>

Recall that you can also access information directly from within IPython by importing the above packages and using IPython's help utility:

```
In [32]: numpy?
```

or

```
In [33]: scipy.special?
```

There is another extremely useful aspect of ufuncs that this recipe did not cover: the important subject of **broadcasting**. We'll take a detailed look at this topic in section X.X.

Aggregations: Min, Max, and Everything In Between

Often when faced with a large amount of data, a first step is to compute summary statistics for the data in question. Perhaps the most common summary statistics are the mean and standard deviation, which allow you to summarize the “typical” values in a dataset, but other aggregates are useful as well (the sum, product, median, min and max, quantiles, etc.)

NumPy has fast built-in aggregation functions for working on arrays; we'll discuss and demonstrate some of them here.

Examples of NumPy Aggregates

Here we'll give a few examples of NumPy's aggregates

Summing the Values in an Array

As a quick example, consider computing the sum of all values in an array. Python itself can do this using the built-in `sum` function

```
import numpy as np

L = np.random.random(100)
sum(L)
47.48211649355256
```

The syntax is quite similar to that of NumPy's `sum` function, and the result is the same.

```
np.sum(L)
47.482116493552553
```

Because the NumPy version executes the operation in compiled code, however, NumPy's version of the operation is computed much more quickly:

```
big_array = np.random.rand(1000000)
%timeit sum(big_array)
%timeit np.sum(big_array)
10 loops, best of 3: 93.4 ms per loop
1000 loops, best of 3: 597 µs per loop
```

Be careful, though: the `sum` function and the `np.sum` function are not identical, which can sometimes lead to confusion! In particular, their optional arguments have different meanings: `np.sum` is aware of multiple array dimensions, as we will see below.

Minimum and Maximum

Similarly, Python has built-in `min` and `max` functions, used to find the minimum value and maximum value of any given array:

```
min(big_array), max(big_array)
(2.3049172436229171e-06, 0.99999789905734671)
```

NumPy's corresponding functions have similar syntax, and again operate much more quickly:

```
np.min(big_array), np.max(big_array)
(2.3049172436229171e-06, 0.99999789905734671)

%timeit min(big_array)
%timeit np.min(big_array)
10 loops, best of 3: 69 ms per loop
1000 loops, best of 3: 444 µs per loop
```

For `min`, `max`, `sum`, and several other NumPy aggregates, a shorter syntax is to use methods of the array object itself:

```
big_array.min(), big_array.max(), big_array.sum()
(2.3049172436229171e-06, 0.99999789905734671, 500500.73894520913)
```

Whenever possible, make sure that you are using the NumPy version of these aggregates when operating on NumPy arrays!

Multi-dimensional Aggregates

One common type of aggregation operation is an aggregate along a row or column. Say you have some data stored in a two-dimensional array:

```
M = np.random.random((3, 4))
print(M)
[[ 0.87592335  0.05691319  0.05410134  0.81738248]
 [ 0.60644247  0.5832756   0.91279565  0.38541931]
 [ 0.89088002  0.68781086  0.52309224  0.71095617]]
```

By default, each NumPy aggregation function will return the aggregate over the entire array:

```
M.sum()
7.1049926640949987
```

Aggregation functions take an additional argument specifying the *axis* along which the aggregate is computed. For example, we can find the minimum value within each column by specifying `axis=0`:

```
M.min(axis=0)
array([ 0.60644247,  0.05691319,  0.05410134,  0.38541931])
```

The function returns four values, corresponding to the four columns of numbers.

Similarly, we can find the maximum value within each row:

```
M.max(axis=1)
array([ 0.87592335,  0.91279565,  0.89088002])
```

The way the axis is specified here can be confusing to users coming from other languages. The `axis` keyword specifies the *dimension of the array which will be collapsed*, rather than the dimension that will be returned. So specifying `axis=0` means that the first axis will be collapsed: for two-dimensional arrays, this means that values within each column will be aggregated.

Other Aggregation Functions

NumPy provides many other aggregation functions which we won't discuss in detail. Additionally, most aggregates have a NaN-safe counterpart, which computes the result while ignoring missing values, which are marked by the special IEEE floating-point NaN value. Some of these NaN-safe functions were added NumPy 1.8-1.9, so they will not be available in older NumPy versions.

The following gives a table of useful aggregation functions available in numpy.

Function Name	NaN-safe Version	Description
<code>np.sum</code>	<code>np.nansum</code>	Compute sum of elements
<code>np.prod</code>	<i>N/A</i>	Compute product of elements
<code>np.mean</code>	<code>np.nanmean</code>	Compute median of elements
<code>np.std</code>	<code>np.nanstd</code>	Compute standard deviation
<code>np.var</code>	<code>np.nanvar</code>	Compute variance
<code>np.min</code>	<code>np.nanmin</code>	Find minimum value
<code>np.max</code>	<code>np.nanmax</code>	Find maximum value
<code>np.argmin</code>	<code>np.nanargmin</code>	Find index of minimum value
<code>np.argmax</code>	<code>np.nanargmax</code>	Find index of maximum value
<code>np.median</code>	<code>np.nanmedian</code>	Compute median of elements
<code>np.percentile</code>	<code>np.nanpercentile</code>	Compute rank-based statistics of elements
<code>np.any</code>	<i>N/A</i>	Evaluate whether any elements are true
<code>np.all</code>	<i>N/A</i>	Evaluate whether all elements are true

We will see these aggregates often throughout the rest of the book.

Example: How Tall is the Average US President?

Aggregates available in NumPy can be extremely useful for summarizing a set of values. As a simple example, let's consider the heights of all United States presidents. We have this data in the file `president_heights.csv`, which is a simple comma-separated list of labels and values:

```
!head -4 president_heights.csv
order,name,height(cm)
1,George Washington,189
2,John Adams,170
3,Thomas Jefferson,189
```

We'll use the Pandas package to read the file and extract this information; further information about Pandas and CSV files can be found in chapter X.X.

```
import pandas as pd
data = pd.read_csv('president_heights.csv')
heights = np.array(data['height(cm)'])
print(heights)
[189 170 189 163 183 171 185 168 173 183 173 175 178 183 193 178 173
 174 183 183 168 170 178 182 180 183 178 182 188 175 179 183 193 182 183
 177 185 188 188 182 185]
```

Note that the heights are measured in centimeters. Now that we have this data array, we can compute a variety of summary statistics (note that all heights are measured in centimeters):

```

print("Mean height:      ", heights.mean())
print("Standard deviation:", heights.std())
print("Minimum height:   ", heights.min())
print("Maximum height:   ", heights.max())
Mean height:      179.738095238
Standard deviation: 6.93184344275
Minimum height:   163
Maximum height:   193

```

Note that in each case, the aggregation operation reduced the entire array to a single summarizing value, which gives us information about the distribution of values. We may also wish to compute quantiles:

```

print("25th percentile:  ", np.percentile(heights, 25))
print("Median:           ", np.median(heights))
print("75th percentile:  ", np.percentile(heights, 75))
25th percentile:   174.25
Median:            182.0
75th percentile:   183.0

```

We see that the median height of US presidents has been 182 cm, or just shy of six feet.

Of course, sometimes it's more useful to see a more visual representation of this data. We can do this using tools in Matplotlib, which will be discussed further in chapter X.X:

```

# See section X.X for a description of these imports
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # set plot style

plt.hist(heights)
plt.title('Height Distribution of US Presidents')
plt.xlabel('height (cm)')
plt.ylabel('number');

```

These aggregates are some of the fundamental pieces of exploratory data analysis that we'll explore in more depth in later sections of the book.

Computation on Arrays: Broadcasting

We saw in the previous section how NumPy's Universal Functions can be used to *vectorize* operations and thereby remove slow Python loops. Another means of vectorizing operations is to use NumPy's *broadcasting* functionality. Broadcasting is simply a set of rules for applying universal functions like addition, subtraction, multiplication, and others on arrays of different sizes.

Introducing Broadcasting

Recall that for arrays of the same size, binary operations are performed on an element-by-element basis:

```
import numpy as np

a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
array([5, 6, 7])
```

Broadcasting allows these types of binary operations to be performed on arrays of different size: for example, we can just as easily add a scalar (think of it as a zero-dimensional array) to an array:

```
a + 5
array([5, 6, 7])
```

We can think of this as an operation that stretches or duplicates the value 5 into the array [5, 5, 5], and adds the results. The advantage of NumPy's broadcasting is that this duplication of values does not actually take place, but it's a useful mental model as we think about broadcasting.

We can similarly extend this to arrays of higher dimension. Observe the result when we add a one-dimensional array to a two-dimensional array:

```
M = np.ones((3, 3))
M
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])

M + a
array([[ 1.,  2.,  3.],
       [ 1.,  2.,  3.],
       [ 1.,  2.,  3.]])
```

Here the one-dimensional array `a` is stretched, or broadcast across the second dimension in order to match the shape of `M`.

While these examples are relatively easy to understand, more complicated cases can involve broadcasting of both arrays. Consider the following example:

```
a = np.arange(3)
b = np.arange(3)[: , np.newaxis]

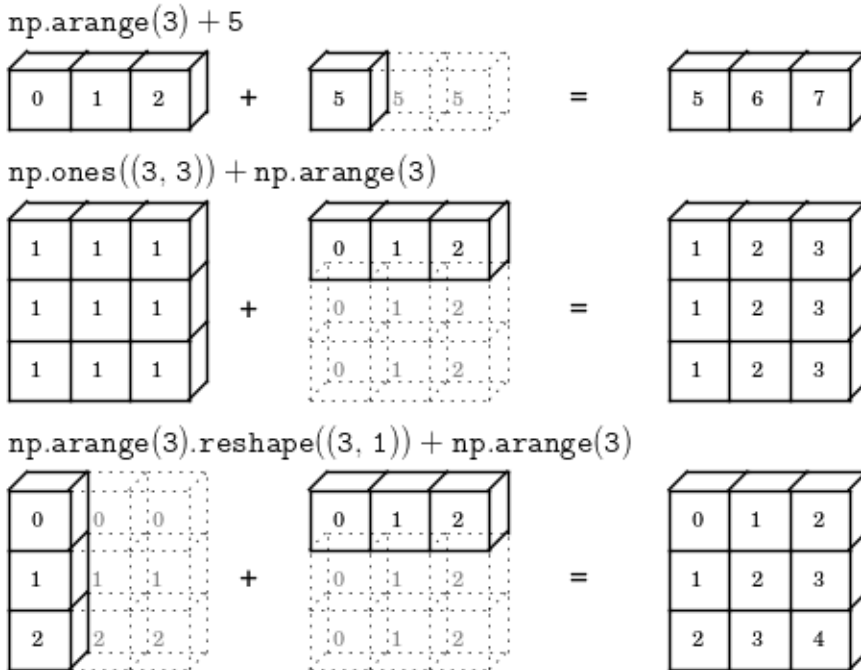
print(a)
print(b)
[0 1 2]
[[0]
 [1]
 [2]]
```

```

a + b
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])

```

Just as above we stretched or broadcasted one value to match the shape of the other, here we've stretched *both* a and b to match a common shape, and the result is a two-dimensional array! The geometry of the above examples is visualized in the following figure:



The dotted boxes represent the broadcasted values: again, this extra memory is not actually allocated in the course of the operation, but it can be useful conceptually to imagine that it is.

Rules of Broadcasting

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

1. If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (left) side.

2. If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
3. If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

To make these rules clear, let's consider a few examples in detail:

Broadcasting Example 1:

Let's look at adding a two-dimensional array to a 1-dimensional array:

```
M = np.ones((2, 3))
a = np.arange(3)

print(M.shape)
print(a.shape)
(2, 3)
(3,)
```

Let's consider an operation on these two arrays. The shape of the arrays are

- M.shape = (2, 3)
- a.shape = (3,)

We see by rule 1 that the array a has fewer dimensions, so we pad it on the left with ones:

- M.shape -> (2, 3)
- a.shape -> (1, 3)

By rule 2, we now see that the first dimension disagrees, so we stretch this dimension to match:

- M.shape -> (2, 3)
- a.shape -> (2, 3)

The shapes match, and we see that the final shape will be (2, 3):

```
M + a
array([[ 1.,  2.,  3.],
       [ 1.,  2.,  3.]])
```

Broadcasting Example 2:

Let's take a look at an example where both arrays need to be broadcast:

```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

Again, we'll start by writing out the shape of the arrays:

- a.shape = (3, 1)

- `b.shape = (3,)`

Rule 1 says we must pad the shape of `b` with ones:

- `a.shape -> (3, 1)`
- `b.shape -> (1, 3)`

And rule 2 tells us that we upgrade each of these ones to match the corresponding size of the other array:

- `a.shape -> (3, 3)`
- `b.shape -> (3, 3)`

Since the result matches, these shapes are compatible. We can see this here:

```
a + b
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Broadcasting Example 3:

Now let's take a look at an example in which the two arrays are not compatible:

```
M = np.ones((3, 2))
a = np.arange(3)

print(M.shape)
print(a.shape)
(3, 2)
(3,)
```

This is just a slightly different situation than in example 1: the matrix `M` is transposed. How does this affect the calculation? The shape of the arrays are

- `M.shape = (3, 2)`
- `a.shape = (3,)`

Again, rule 1 tells us that we must pad the shape of `a` with ones:

- `M.shape -> (3, 2)`
- `a.shape -> (1, 3)`

By rule 2, the first dimension of `a` is stretched to match that of `M`:

- `M.shape -> (3, 2)`
- `a.shape -> (3, 3)`

Now we hit rule 3: the final shapes do not match, so these two arrays are incompatible, as we can observe by attempting this operation:

```
M + a
```

Note the potential confusion here: you could imagine making `a` and `M` compatible by, say, padding the size of `a` with ones on the right rather than the left. But this is not how the broadcasting rules work! That sort of flexibility might be useful in some cases, but it would lead to potential areas of ambiguity. If right-side-padding is what you'd like, you can do this explicitly by reshaping the array (we'll use the `np.newaxis` keyword that we introduce in section X.X):

```
a[:, np.newaxis].shape
(3, 1)

M + a[:, np.newaxis]
array([[ 1.,  1.],
       [ 2.,  2.],
       [ 3.,  3.]])
```

Also note that while we've been focusing on the `+` operator here, these broadcasting rules apply to *any* binary ufunc. For example, here is the `logaddexp(a, b)` function, which computes `log(exp(a) + exp(b))` with more precision than the naive approach

```
np.logaddexp(M, a[:, np.newaxis])
array([[ 1.31326169,  1.31326169],
       [ 1.69314718,  1.69314718],
       [ 2.31326169,  2.31326169]])
```

For more information on the many available universal functions, refer to section X.X.

Broadcasting in Practice

Broadcasting operation form the core of many examples we'll see throughout this book; here are a couple simple examples of where they can be useful:

Centering an Array

In the previous section we saw that ufuncs allow a NumPy user to remove the need to explicitly write slow Python loops. Broadcasting extends this ability. One commonly-seen example is when centering an array of data. Imagine you have an array of ten observations, each of which consists of three values. Using the standard convention, we'll store this in a 10×3 array:

```
X = np.random.random((10, 3))
```

We can compute the mean using the `mean` aggregate across the first dimension:

```
Xmean = X.mean(0)
Xmean
array([ 0.54579044,  0.61639938,  0.51815359])
```

And now we can center the X array by subtracting the mean (this is a broadcasting operation):

```
X_centered = X - Xmean
```

To double-check that we've done this correctly, we can check that the centered array has near zero mean:

```
X_centered.mean(0)
array([-1.11022302e-17,  6.66133815e-17, -8.88178420e-17])
```

To within machine precision, the mean is now zero.

Plotting a 2D function

One place that broadcasting is very useful is in displaying images based on 2D functions. If we want to define a function $z = f(x, y)$, broadcasting can be used to compute the function across the grid:

```
# x and y have 50 steps from 0 to 5
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 50)[: , np.newaxis]

z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)

# Use of matplotlib is discussed further in section X.X
%matplotlib inline
import matplotlib.pyplot as plt

plt.imshow(z, origin='lower', extent=[0, 5, 0, 5],
           cmap=plt.cm.cubehelix)
plt.colorbar();
```

More details on creating graphics with matplotlib can be found in Chapter X.X.

Utility Routines for Broadcasting

`np.broadcast` and `np.broadcast_arrays` are utility routines which can help with broadcasting.

`np.broadcast` will create a broadcast object which gives the shape of the broadcasted arrays, and allows iteration through all broadcasted sets of elements:

```
x = [['A'],
      ['B']]
y = [[1, 2, 3]]
xy = np.broadcast(x, y)

xy.shape
(2, 3)

for xi, yi in xy:
    print(xi, yi)
A 1
```

```
A 2
A 3
B 1
B 2
B 3
```

`np.broadcast_arrays` takes input arrays and returns array views with the resulting shape and structure:

```
xB, yB = np.broadcast_arrays(x, y)
print(xB)
print(yB)
[['A' 'A' 'A']
 ['B' 'B' 'B']]
[[1 2 3]
 [1 2 3]]
```

These two functions will prove useful when working with arrays of different sizes; we will occasionally see them in action through the remainder of this text.

Comparisons, Masks, and Boolean Logic

This section covers the use of Boolean masks to examine and manipulate values within NumPy arrays. Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion: for example, you might wish to count all values greater than a certain value, or perhaps remove all outliers which are above some threshold. In NumPy, boolean masking is often the most efficient way to accomplish these types of tasks.

Example: Counting Rainy Days

Imagine you have a series of data that represents the amount of precipitation each day for a year in a given city. For example, here we'll load the daily rainfall statistics for the city of Seattle in 2014, using the pandas tools covered in more detail in section X.X:

```
import numpy as np
import pandas as pd

# use Pandas extract rainfall inches as a NumPy array
rainfall = pd.read_csv('Seattle2014.csv')['PRCP'].values
inches = rainfall / 254 # 1/10mm -> inches
inches.shape
(365,)
```

The array contains 365 values, giving daily rainfall in inches from January 1 to December 31, 2014.

As a first quick visualization, Let's look at the histogram of rainy days:

```
# More information on matplotlib can be found in Chapter X.X
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # set plot styles

plt.hist(inches, 40);
```

This histogram gives us a general idea of what the data looks like: the vast majority of days in Seattle in 2014 (despite its reputation) saw near zero measured rainfall. But this doesn't do a good job of conveying some information we'd like to see: for example, how many rainy days were there in the year? What is the average precipitation on those rainy days? How many days were there with more than half an inch of rain?

Digging Into the Data

One approach to this would be to answer these questions by hand: loop through the data, incrementing a counter each time we see values in some desired range. For reasons discussed through this chapter, such an approach is very inefficient: both from the standpoint of both time writing code and time computing the result. We saw in section X.X that NumPy's *Universal Functions* can be used in place of loops to do fast elementwise arithmetic operations on arrays; in the same way, we can use other ufuncs to do elementwise *comparisons* over arrays, and we can then manipulate the results to answer the questions we have. We'll leave the data aside for right now, and discuss some general tools in NumPy to use *masking* to quickly answer these types of questions.

Comparison Operators as ufuncs

In section X.X we introduced NumPy's Universal Functions (ufuncs), and focused in particular on arithmetic operators. We saw that using `+`, `-`, `*`, `/`, and others on arrays leads to *elementwise* operations. For example, adding a number to an array adds that number to every element:

```
x = np.array([1, 2, 3, 4, 5])
x + 10
array([11, 12, 13, 14, 15])
```

NumPy also implements comparison operators such as `<` (less than) and `>` (greater than) as elementwise ufuncs. The result of these comparison operators is always an array with a boolean data type. All six of the standard comparison operations are available:

```
x < 3 # less than
array([ True,  True, False, False, False], dtype=bool)

x > 3 # greater than
array([False, False, False,  True,  True], dtype=bool)

x <= 3 # less than or equal
array([ True,  True,  True, False, False], dtype=bool)
```

```

x >= 3 # greater than or equal
array([False, False,  True,  True], dtype=bool)

x != 3 # not equal
array([  True,  True, False,  True], dtype=bool)

x == 3 # equal
array([False, False,  True, False], dtype=bool)

```

It is also possible to compare two arrays element-by-element, and to include compound expressions:

```

2 * x == x ** 2
array([False,  True, False, False], dtype=bool)

```

As in the case of arithmetic operators, the comparison operators are implemented as universal functions in NumPy; for example, when you write `x < 3`, internally NumPy uses `np.less(x, 3)`. A summary of the comparison operators and their equivalent ufunc is shown below:

Operator	Equivalent ufunc
<code>==</code>	<code>np.equal</code>
<code>!=</code>	<code>np.not_equal</code>
<code><</code>	<code>np.less</code>
<code><=</code>	<code>np.less_equal</code>
<code>></code>	<code>np.greater</code>
<code>>=</code>	<code>np.greater_equal</code>

Just as in the case of arithmetic ufuncs, these will work on arrays of any size and shape. Here is a two-dimensional example:

```

rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
x
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])

x < 6
array([[ True,  True,  True,  True],
       [False, False,  True,  True],
       [ True,  True, False, False]], dtype=bool)

```

In each case, the result is a boolean array. Working with boolean arrays is straightforward, and there are a few common patterns that we'll mention here:

Working with Boolean Arrays

Given a boolean array, there are a host of useful operations you can do. We'll work with `x`, the two-dimensional array we created above.

```
print(x)
[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
```

Counting Entries

To count the number of `True` entries in a boolean array, `np.count_nonzero` is useful:

```
# how many values less than six?
np.count_nonzero(x < 6)
8
```

We see that there are eight array entries that are less than six. Another way to get at this information is to use `np.sum`; in this case `False` is interpreted as 0, and `True` is interpreted as 1:

```
np.sum(x < 6)
8
```

The benefit of `sum()` is that like with other NumPy aggregation functions, this summation can be done along rows or columns as well:

```
# how many values less than six in each row?
np.sum(x < 6, 1)
array([4, 2, 2])
```

This counts the number of values less than six in each row of the matrix.

If we're interested in quickly checking whether any or all the values are true, we can use (you guessed it) `np.any` or `np.all`:

```
# are there any values greater than eight?
np.any(x > 8)
True

# are there any values less than zero?
np.any(x < 0)
False

# are all values less than ten?
np.all(x < 10)
True

# are all values equal to six?
np.all(x == 6)
False
```

`np.all` and `np.any` can be used along particular axes as well. For example:

```
# are all values in each row less than four?
np.all(x < 8, axis=1)
array([ True, False,  True], dtype=bool)
```

Here all the elements in the first and third rows are less than eight, while this is not the case for the second row.

Finally, a quick warning: as mentioned in Section X.X, Python has built-in `sum()`, `any()`, and `all()` functions. These have a different syntax than the NumPy versions, and in particular will fail or produce unintended results when used on multi-dimensional arrays. Be sure that you are using `np.sum()`, `np.any()`, and `np.all()` for these examples!

Boolean Operators

Above we saw how to count, say, all days with rain less than four inches, or all days with rain greater than two inches. But what if we want to know about all days with rain less than four inches AND greater than one inch? This is accomplished through Python's *bitwise logic operators*, `&`, `|`, `^`, and `~`, first discussed in Section X.X. Like with the standard arithmetic operators, NumPy overloads these as ufuncs which work element-wise on (usually boolean) arrays.

For example, we can address this sort of compound question this way:

```
np.sum((inches > 0.5) & (inches < 1))
29
```

So we see that there are 29 days with rainfall between 0.5 and 1.0 inches.

Note that the parentheses here are important: because of operator precedence rules, with parentheses removed this expression would be evaluated as

```
inches > (1 & inches) < 4
```

which results in an error.

Using boolean identities, we can answer questions in terms of other boolean operators. Here, we answer the same question in a more convoluted way, using boolean identities:

```
np.sum(~( (inches <= 0.5) | (inches >= 1) ))
29
```

As you can see, combining comparison operators and boolean operators on arrays can lead to a wide range of possible logical operations on arrays.

The following table summarizes the bitwise boolean operators and their equivalent ufuncs:

Operator	Equivalent ufunc
&	np.bitwise_and
	np.bitwise_or
^	np.bitwise_xor
~	np.bitwise_not

Returning to Seattle's Rain

With these tools in mind, we can start to answer the types of questions we have about this data. Here are some examples of results we can compute when combining masking with aggregations:

```
print("Number days without rain:      ", np.sum(inches == 0))
print("Number days with rain:        ", np.sum(inches != 0))
print("Days with more than 0.5 inches:", np.sum(inches > 0.5))
print("Days with 0.2 to 0.5 inches:  ", np.sum((inches > 0.2) & (inches <
0.5)))
Number days without rain:      215
Number days with rain:        150
Days with more than 0.5 inches: 37
Days with 0.2 to 0.5 inches:   36
```

Boolean Arrays as Masks

Above we looked at aggregates computed directly on boolean arrays. A more powerful pattern is to use boolean arrays as masks, to select particular subsets of the data themselves. Returning to our `x` array from above, suppose we want an array of all values in the array which are less than, say, 5.

```
x
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])
```

We can obtain a boolean array for this condition easily, as we saw above:

```
x < 5
array([[False,  True,  True,  True],
       [False, False,  True, False],
       [ True,  True, False, False]], dtype=bool)
```

Now to *select* these values from the array, we can simply index on this boolean array: this is known as a *masking* operation:

```
x[x < 5]
array([0, 3, 3, 3, 2, 4])
```

What is returned is a one-dimensional array filled with all the values which meet this condition; in other words, all the values in positions at which the mask array is `True`.

We are then free to operate on these values as we wish. For example, we can compute some relevant statistics on the data:

```
# construct a mask of all rainy days
rainy_days = (inches > 0)

# construct a mask of all summer days (June 21st is the 172nd day)
summer = (np.arange(365) - 172 < 90)

print("Median daily precip on rainy days in 2014 (inches):",
      np.median(inches[rainy_days]))
print("Median daily precip overall, summer 2014 (inches):",
      np.median(inches[summer]))
print("Maximum daily precip, summer 2014 (inches):", np.max(inches[summer]))
print("Median precip on all non-summer rainy days (inches):",
      np.median(inches[rainy_days & ~summer]))
Median daily precip on rainy days in 2014 (inches): 0.194881889764
Median daily precip overall, summer 2014 (inches): 0.0
Maximum daily precip, summer 2014 (inches): 1.83858267717
Median precip on all non-summer rainy days (inches): 0.224409448819
```

By combining boolean operations, masking operations, and aggregates, we can very quickly answer these sorts of questions for our dataset.

Sidebar: "&" vs. "and"...

One common point of confusion is the difference between the keywords "and" and "or"; and the operators & and |. When would you use one versus the other?

The difference is this: "and" and "or" gauge to the truth or falsehood of *entire object*, while & and | refer to *portions of each object*.

When you use "and" or "or", it's equivalent to asking Python to treat the object as a single boolean entity. In Python, all nonzero integers will evaluate as True. Thus:

```
bool(42), bool(27)
(True, True)

bool(42 and 27)
True

bool(42 or 27)
True
```

When you use & and | on integers, the expression operates on the bits of the element, applying the *and* or the *or* to the individual bits making up the number:

```
print(bin(42))
print(bin(59))
0b101010
0b111011
```

```
print(bin(42 & 59))
0b101010

bin(42 | 59)
'0b111011'
```

Notice that the corresponding bits (right to left) of the binary representation are compared in order to yield the result.

When you have an array of boolean values in NumPy, this can be thought of as a string of bits where 1 = True and 0 = False, and the result of & and | operates similarly to above:

```
A = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
B = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
A | B
array([ True,  True,  True, False,  True,  True], dtype=bool)
```

Using or on these arrays will try to evaluate the truth or falsehood of the entire array object, which is not a well-defined value:

```
A or B
```

Similarly, when doing a boolean expression on a given array, you should use | or & rather than or or and:

```
x = np.arange(10)
(x > 4) & (x < 8)
array([False, False, False, False,  True,  True,  True, False, False],
      dtype=bool)
```

Trying to evaluate the truth or falsehood of the entire array will give the same ValueError we saw above:

```
(x > 4) and (x < 8)
```

So remember this: "and" and "or" perform a single boolean evaluation on an entire object, while & and | perform multiple boolean evaluations on the content (the individual bits or bytes) of an object. For boolean NumPy arrays, the latter is nearly always the desired operation.

Fancy Indexing

In the previous section we saw how to access and modify portions of arrays using simple indices (e.g. `arr[0]`), slices (e.g. `arr[:5]`), and boolean masks (e.g. `arr[arr > 0]`). In this section we'll look at another style of array indexing, known as *fancy indexing*. Fancy indexing is like the simple indexing above, but we pass arrays of indices in place of single scalars. This allows us to very quickly access and modify complicated subsets of an array's values.

Exploring Fancy Indexing

Fancy indexing is conceptually simple: it simply means passing an array of indices to access multiple array elements at once. For example, consider the following array:

```
import numpy as np
rand = np.random.RandomState(42)

x = rand.randint(100, size=10)
print(x)
[51 92 14 71 60 20 82 86 74 74]
```

Suppose we want to access three different elements. We could do it like this:

```
x[3], x[7], x[2]
[71, 86, 14]
```

Alternatively, we can pass a single list or array of indices to obtain the same result:

```
ind = [3, 7, 4]
x[ind]
array([71, 86, 60])
```

When using fancy indexing, the shape of the result reflects the shape of the *index arrays* rather than the shape of the *array being indexed*:

```
ind = np.array([[3, 7],
                [4, 5]])
x[ind]
array([[71, 86],
       [60, 20]])
```

Fancy indexing also works in multiple dimensions. Consider the following array:

```
X = np.arange(12).reshape((3, 4))
X
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Like with standard indexing, the first index refers to the row, and the second to the column:

```
row = np.array([0, 1, 2])
col = np.array([2, 1, 3])
X[row, col]
array([ 2,  5, 11])
```

Notice that the first value in the result is $X[0, 2]$, the second is $X[1, 1]$, and the third is $X[2, 3]$. The pairing of indices in fancy indexing is even more powerful than this: it follows all the broadcasting rules that were mentioned in section X.X. So, for example, if we combine a column vector and a row vector within the indices, we get a two-dimensional result:

```
X[row[:, np.newaxis], col]
array([[ 2,  1,  3],
       [ 6,  5,  7],
       [10,  9, 11]])
```

Here, each row value is matched with each column vector, exactly as we saw in broadcasting of arithmetic operations. For example:

```
row[:, np.newaxis] * col
array([[0, 0, 0],
       [2, 1, 3],
       [4, 2, 6]])
```

It is always important to remember with fancy indexing that the return value reflects the *shape of the indices*, rather than the shape of the array being indexed.

Combined Indexing

For even more powerful operations, fancy indexing can be combined with the other indexing schemes we've seen.

```
print(X)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

We can combine fancy and simple indices:

```
X[2, [2, 0, 1]]
array([10,  8,  9])
```

We can also combine fancy indexing with slicing:

```
X[1:, [2, 0, 1]]
array([[ 6,  4,  5],
       [10,  8,  9]])
```

And we can combine fancy indexing with masking:

```
mask = np.array([1, 0, 1, 0], dtype=bool)
X[row[:, np.newaxis], mask]
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
```

All of these indexing options combined lead to a very flexible set of operations for accessing and modifying array values.

Generating Indices: `np.where`

One commonly-seen pattern is to use `np.where` (or the very similar `np.nonzero`) to generate indices to use within fancy indexing. We saw previously that you can use

boolean masks to select certain elements of an array. Here, let's create a random array and select all the even elements:

```
X = rand.randint(10, size=(3, 4))
X
array([[7, 4, 3, 7],
       [7, 2, 5, 4],
       [1, 7, 5, 1]])

evens = X[X % 2 == 0]
evens
array([4, 2, 4])
```

Equivalently, you might use the `np.where` function:

```
X[np.where(X % 2 == 0)]
array([4, 2, 4])
```

What does `np.where` do? In this case, we have given it a boolean mask, and it has returned a set of indices:

```
i, j = np.where(X % 2 == 0)
print(i)
print(j)
[0 1 1]
[1 1 3]
```

These indices, like the ones we saw above, are interpreted in pairs: (i.e. the first element is `X[0, 1]`, the second is `X[0, 2]`, etc.) Note here that the computation of these indices is an extra step, and thus using `np.where` in this manner will generally be less efficient than simply using the boolean mask itself. So why might you use `np.where`? Many use it because they have come from a language like IDL or MatLab where such constructions are familiar. But `np.where` can be useful in itself when the indices themselves are of interest, and also has other functionality which you can read about in its documentation.

Example: Selecting Random Points

One common use of fancy indexing is the selection of subsets of rows from a matrix. By convention, the values of N points in D dimensions are often represented by a 2-dimensional, $N \times D$ array. We'll generate some points from a two-dimensional multivariate normal distribution:

```
mean = [0, 0]
cov = [[1, 2],
       [2, 5]]
X = rand.multivariate_normal(mean, cov, 100)
X.shape
(100, 2)
```

Using the plotting tools we will discuss in chapter X.X, we can visualize these points:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # for plot styling
```

```
plt.scatter(X[:, 0], X[:, 1]);
```

Let's use fancy indexing to select 20 random points. We'll do this by first choosing 20 random indices with no repeats, and use these indices to select a portion of the original array:

```
indices = np.random.choice(X.shape[0], 20, replace=False)
selection = X[indices] # fancy indexing here
selection.shape
(20, 2)
```

Now to see which points were selected, let's over-plot large circles at the locations of the selected points:

```
plt.scatter(X[:, 0], X[:, 1], alpha=0.3)
plt.scatter(selection[:, 0], selection[:, 1],
            facecolor='none', s=200);
```

This sort of strategy is often used to quickly partition datasets, as is often needed in train/test splitting for validation of statistical models (see section X.X). We'll see further uses of fancy indexing throughout the book. More information on plotting with matplotlib is available in chapter X.X.

Modifying values with Fancy Indexing

Above we saw how to access parts of an array with fancy indexing. Fancy indexing can also be used to modify parts of an array.

For example, imagine we have an array of indices and we'd like to set the corresponding items in an array to some value:

```
x = np.arange(10)
i = np.array([2, 1, 8, 4])
x[i] = 99
print(x)
[ 0 99 99  3 99  5  6  7 99  9]
```

We can use any assignment-type operator for this. For example:

```
x[i] -= 10
print(x)
[ 0 89 89  3 89  5  6  7 89  9]
```

Notice, though, that repeated indices with these operations can cause some potentially unexpected results. Consider the following:

```
x = np.zeros(10)
x[[0, 0]] = [4, 6]
```

```
print(x)
[ 6.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

Where did the 4 go? The result of this operation is to first assign $x[0] = 4$, followed by $x[0] = 6$. The result, of course, is that $x[0]$ contains the value 6.

Fair enough, but consider this operation:

```
i = [2, 3, 3, 4, 4, 4]
x[i] += 1
x
array([ 6.,  0.,  1.,  1.,  1.,  0.,  0.,  0.,  0.,  0.])
```

You might expect that $x[3]$ would contain the value 2, and $x[3]$ would contain the value 3, as this is how many times each index is repeated. Why is this not the case? Conceptually, this is because $x[i] += 1$ is meant as a short-hand of $x[i] = x[i] + 1$. $x[i] + 1$ is evaluated, and then the result is assigned to the indices in x . With this in mind, it is not the augmentation that happens multiple times, but the assignment, which leads to the rather non-intuitive results.

So what if you want the other behavior where the operation is repeated? For this, you can use the `at()` method of ufuncs (available since NumPy 1.8), and do the following:

```
x = np.zeros(10)
np.add.at(x, i, 1)
print(x)
[ 0.  0.  1.  2.  3.  0.  0.  0.  0.  0.]
```

The `at()` method does an in-place application of the given operator at the specified indices (here, `i`) with the specified value (here, 1). Another method which is similar in spirit is the `reduceat()` method of ufuncs, which you can read about in the NumPy documentation.

Example: Binning data

You can use these ideas to quickly bin data to create a histogram. For example, imagine we have 1000 values and would like to quickly find where they fall within an array of bins. We could compute it using `ufunc.at` like this:

```
np.random.seed(42)
x = np.random.randn(100)

# compute a histogram by-hand
bins = np.linspace(-5, 5, 20)
counts = np.zeros_like(bins)

# find the appropriate bin for each x
i = np.searchsorted(bins, x)

# add 1 to each of these bins
np.add.at(counts, i, 1)
```


You might notice that what we've done is to simply compute the contents of a histogram:

```
# plot the results
plt.plot(bins, counts, linestyle='steps');
```

Of course, it would be silly to have to do this each time you want to plot a histogram. This is why matplotlib provides the `plt.hist()` routine which does the same in a single line:

```
plt.hist(x, bins, histtype='step');
```

To compute the binning, matplotlib uses the `np.histogram` function, which does a very similar computation to what we did above. Let's compare the two here:

```
print("NumPy routine:")
%timeit counts, edges = np.histogram(x, bins)

print("Custom routine:")
%timeit np.add.at(counts, np.searchsorted(bins, x), 1)
NumPy routine:
The slowest run took 4.70 times longer than the fastest. This could mean that
an intermediate result is being cached
10000 loops, best of 3: 69.6 µs per loop
Custom routine:
100000 loops, best of 3: 19.2 µs per loop
```

Our own one-line algorithm is several times faster than the optimized algorithm in NumPy! How can this be? If you dig into the `np.histogram` source code (you can do this in IPython by typing `np.histogram??`) you'll see that it's quite a bit more involved than the simple search-and-count that we've done: this is because NumPy's algorithm is more flexible, and particularly is designed for better performance when the number of data points becomes large:

```
x = np.random.randn(1000000)
print("NumPy routine:")
%timeit counts, edges = np.histogram(x, bins)

print("Custom routine:")
%timeit np.add.at(counts, np.searchsorted(bins, x), 1)
NumPy routine:
10 loops, best of 3: 65.1 ms per loop
Custom routine:
10 loops, best of 3: 129 ms per loop
```

What this comparison shows is that algorithmic efficiency is almost never a simple question. An algorithm efficient for large datasets will not always be the best choice for small datasets, and vice versa (see *big-O notation* in section X.X). But the advantage of coding this algorithm yourself is that with an understanding of these basic methods, you could use these building blocks to extend this to do some very interesting custom behaviors. The key to efficiently using Python in data-intensive applica-

tions is knowing about general convenience routines like `np.histogram` and when they're appropriate, but also knowing how to implement your own efficient custom algorithms for times that you need more pointed behavior.

Numpy Indexing Tricks

This section goes over some of NumPy's *indexing tricks*. They are quick and powerful constructs to build certain types of arrays very quickly and easily, but the very terseness which makes them useful can make them difficult to understand. For this reason, even many experienced NumPy users have never used or even seen these; for most use cases, I'd recommend avoiding these in favor of more standard indexing operations. Nevertheless, this is an interesting enough topic that we'll cover it briefly here, if only to give you a means to brazenly confuse and befuddle your collaborators.

All the constructs below come from the module `numpy.lib.index_tricks`. Here is a listing of the tricks we'll go over:

- `numpy.mgrid`: construct dense multi-dimensional mesh grids
- `numpy.ogrid`: construct open multi-dimensional mesh grids
- `numpy.ix_`: construct an open multi-index grid
- `numpy.r_`: translate slice objects to concatenations along rows
- `numpy.c_`: translate slice objects to concatenations along columns

All of these index tricks are marked by the fact that rather than providing an interface through function calls (e.g. `func(x, y)`), they provide an interface through indexing and slicing syntax (e.g. `func[x, y]`). This type of re-purposing of slicing syntax is, from what I've seen, largely unique in the Python world, and is why some purists would consider these tricks dirty hacks which should be avoided at all costs. Consider yourself warned.

Because these tricks are a bit overly terse and uncommon, we'll also include some recommendations for how to duplicate the behavior of each with more commonly-seen and easy to read NumPy constructions. Some functionality with a similar spirit is provided by the objects `numpy.s_` and `numpy.index_exp`: these are primarily useful as utility routines which convert numpy-style indexing into tuples of Python `slice` objects which can then be manipulated independently. We won't cover these two utilities here: for more information refer to the NumPy documentation.

`np.mgrid`: Convenient Multi-dimensional Mesh Grids

The primary use of `np.mgrid` is the quick creation multi-dimensional grids of values. For example, say you want to visualize the function

$$f(x, y) = \text{sinc}(x^2 + y^2)$$

Where $\text{sinc}(x) = \sin(x)/x$. To plot this, we'll first need to create two two-dimensional grids of values for x and y . Using common NumPy broadcasting constructs, we might do something like this:

```
import numpy as np

# Create the 2D arrays
x = np.zeros((3, 4), int)
y = x.copy()

# Fill the arrays with values
x += np.arange(3)[: , np.newaxis]
y += np.arange(4)

print(x)
print(y)
[[0 0 0 0]
 [1 1 1 1]
 [2 2 2 2]]
[[0 1 2 3]
 [0 1 2 3]
 [0 1 2 3]]
```

With `np.mgrid`, we can do the same thing in one line using a slice syntax

```
x, y = np.mgrid[0:3, 0:4]
print(x)
print(y)
[[0 0 0 0]
 [1 1 1 1]
 [2 2 2 2]]
[[0 1 2 3]
 [0 1 2 3]
 [0 1 2 3]]
```

Notice what this object does: it converts a slice syntax (such as `[0:3]`) into a range syntax (such as `range(0, 3)`) and automatically fills the results in a multi-dimensional array. The x value increases along the first axis, while the y value increases along the second axis.

Above we used the default step of 1, but we could just as well use a different step size:

```
x, y = np.mgrid[-2:2:0.1, -2:2:0.1]
print(x.shape, y.shape)
(40, 40) (40, 40)
```

The third slice value gives the step size for the range: 40 even steps of 0.1 between -2 and 2.

Deeper down the rabbit hole: imaginary steps

`np.mgrid` would be useful enough if it stopped there. But what if you prefer to use `np.linspace` rather than `np.arange`? That is, what if you'd like to specify not the step size, but the *number* of steps between the start and end points? A drawn-out way of doing this might look as follows:

```
# Create the 2D arrays
x = np.zeros((3, 5), float)
y = x.copy()

# Fill the arrays with values
x += np.linspace(-1, 1, 3)[: , np.newaxis]
y += np.linspace(-1, 1, 5)

print(x)
print(y)
[[-1. -1. -1. -1. -1.]
 [ 0.  0.  0.  0.  0.]
 [ 1.  1.  1.  1.  1.]]
[[-1. -0.5  0.  0.5  1. ]
 [-1. -0.5  0.  0.5  1. ]
 [-1. -0.5  0.  0.5  1. ]]
```

`np.mgrid` allows you to specify these `linspace` arguments by passing *imaginary* values to the slice. If the third value is an imaginary number, the integer part of the imaginary component will be interpreted as a number of steps. Recalling that the imaginary values in Python are expressed by appending a `j`, we can write:

```
x, y = np.mgrid[-1:1:3j, -1:1:5j]
print(x)
print(y)
[[-1. -1. -1. -1. -1.]
 [ 0.  0.  0.  0.  0.]
 [ 1.  1.  1.  1.  1.]]
[[-1. -0.5  0.  0.5  1. ]
 [-1. -0.5  0.  0.5  1. ]
 [-1. -0.5  0.  0.5  1. ]]
```

Using this now, we can use `np.mgrid` to quickly build-up a grid of values for plotting our 2D function:

```
# grid of 50 steps from -3 to 3
x, y = np.mgrid[-3: 3: 50j,
                 -3: 3: 50j]
f = np.sinc(x ** 2 + y ** 2)
```

We'll plot this with a contour function; for more information on contour plots, see section X.X:

```
%matplotlib inline
import matplotlib.pyplot as plt
```

```
plt.contourf(x, y, f, 100, cmap='cubehelix')
plt.colorbar();
```

The Preferred Alternative: `np.meshgrid`

Because of the non-standard slicing syntax of `np.mgrid`, it should probably not be your go-to method in code that other people will read and use. A more readable alternative is to use the `np.meshgrid` function: though it's a bit less concise, it's much easier for the average reader of your code to understand what's happening.

```
x, y = np.meshgrid(np.linspace(-1, 1, 3),
                   np.linspace(-1, 1, 5), indexing='ij')

print(x)
print(y)
[[-1. -1. -1. -1. -1.]
 [ 0.  0.  0.  0.  0.]
 [ 1.  1.  1.  1.  1.]]
[[-1. -0.5  0.  0.5  1. ]
 [-1. -0.5  0.  0.5  1. ]
 [-1. -0.5  0.  0.5  1. ]]
```

As we see, `np.meshgrid` is a functional interface that takes any two sequences as input. The `indexing` keyword specifies whether the `x` values will increase along columns or rows. This form is much more clear than the rather obscure `np.mgrid` shortcut, and is a good balance between clarity and brevity.

Though we've stuck with two-dimensional examples here, both `mgrid` and `meshgrid` can be used for any number of dimensions: in either case, simply add more range specifiers to the argument list.

`np.ogrid`: Convenient Open Grids

`ogrid` has a very similar syntax to `mgrid`, except it doesn't fill-in the full multi-dimensional array of repeated values. Recall that `mgrid` gives multi-dimensional outputs for each input:

```
x, y = np.mgrid[-1:2, 0:3]
print(x)
print(y)
[[-1 -1 -1]
 [ 0  0  0]
 [ 1  1  1]]
[[0 1 2]
 [0 1 2]
 [0 1 2]]
```

The corresponding function call with `ogrid` returns a simple column and row array:

```
x, y = np.ogrid[-1:2, 0:3]
print(x)
```

```

print(y)
[[-1]
 [ 0]
 [ 1]]
[[0 1 2]]

```

This is useful, because often you can use broadcasting tricks (see section X.X) to obviate the need for the full, dense array. Especially for large arrays, this can save a lot of memory overhead within your calculation.

For example, because `plt.imshow` does not require the dense `x` and `y` grids, if we wanted to use it rather than `plt.contourf` to visualize the 2-dimensional sinc function, we could use `np.ogrid`. NumPy broadcasting takes care of the rest:

```

# grid of 50 steps from -3 to 3
x, y = np.ogrid[-3: 3: 50j,
                -3: 3: 50j]
f = np.sinc(x ** 2 + y ** 2)
plt.imshow(f, cmap='cubehelix',
           extent=[-3, 3, -3, 3])
plt.colorbar();

```

The preferred alternative: Manual reshaping

Like `mgrid`, `ogrid` can be confusing for somebody reading through code. For this type of operation, I prefer using `np.newaxis` to manually reshape input arrays. Compare the following:

```

x, y = np.ogrid[-1:2, 0:3]
print(x)
print(y)
[[-1]
 [ 0]
 [ 1]]
[[0 1 2]]

x = np.arange(-1, 2)[:, np.newaxis]
y = np.arange(0, 3)[np.newaxis, :]
print(x)
print(y)
[[-1]
 [ 0]
 [ 1]]
[[0 1 2]]

```

The average reader of your code is much more likely to have encountered `np.newaxis` than to have encountered `np.ogrid`.

np.ix_: Open Index Grids

`np.ix_` is an index trick which can be used in conjunction with Fancy Indexing (see section X.X). Functionally, it is very similar to `np.ogrid` in that it turns inputs into a multi-dimensional open grid, but rather than generating sequences of numbers based on the inputs, it simply reshapes the inputs:

```
i, j = np.ix_([0, 1], [2, 4, 3])
print(i)
print(j)
[[0]
 [1]]
[[2 4 3]]
```

The result of `np.ix_` is most often used directly as a fancy index. For example:

```
M = np.arange(16).reshape((2, 8))
print(M)
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]]

M[np.ix_([0, 1], [2, 4, 3])]
array([[ 2,  4,  3],
       [10, 12, 11]])
```

Notice what this did: it created a new array with rows specified by the first argument, and columns specified by the second. Like `mgrid` and `ogrid`, `ix_` can be used in any number of dimensions. Often, however, it can be cleaner to specify the indices directly. For example, to find the equivalent of the above result, we can alternatively mix slicing and fancy indexing to write

```
M[:, [2, 4, 3]]
array([[ 2,  4,  3],
       [10, 12, 11]])
```

For more complicated operations, we might instead follow the strategy above under `np.ogrid` and use a solution based on `np.newaxis`.

np.r_: concatenation along rows

`np.r_` is an index trick which allows concise concatenations of arrays. For example, imagine that we want to create an array of numbers which counts up to a value and then back down. We might use `np.concatenate` as follows:

```
np.concatenate([range(5), range(5, -1, -1)])
array([0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0])
```

`np.r_` allows us to do this concisely using index notations similar to those in `np.mgrid` and `np.ogrid`:

```
np.r_[5, 5:-1:-1]
array([0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0])
```

Furthermore, slice notation (with the somewhat confusing imaginary steps discussed above) can be mixed with arrays and lists to create even more flexible concatenations:

```
np.r_[0:1:3j, 3*[5], range(4)]
array([ 0. , 0.5, 1. , 5. , 5. , 5. , 0. , 1. , 2. , 3. ])
```

To make things even more confusing, if the first index argument is a string, it specifies the axis of concatenation. For example, for a two-dimensional array, the string “0” or “1” will indicate whether to stack horizontally or vertically:

```
x = np.array([[0, 1, 2],
              [3, 4, 5]])
np.r_["0", x, x]
array([[0, 1, 2],
       [3, 4, 5],
       [0, 1, 2],
       [3, 4, 5]])
np.r_["1", x, x]
array([[0, 1, 2, 0, 1, 2],
       [3, 4, 5, 3, 4, 5]])
```

Even more complicated options are available for the initial string argument. For example, we can turn one-dimensional arrays into two-dimensional arrays by putting another argument within the string:

```
np.r_["0,2", :3, 1:2:3j, [3, 4, 3]]
array([[ 0. , 1. , 2. ],
       [ 1. , 1.5, 2. ],
       [ 3. , 4. , 3. ]])
```

Roughly, this string argument says “make sure the arrays are two-dimensional, and then concatenate them along axis zero”.

Even more complicated axis specifications are available: you can refer to the documentation of `np.r_` for more information.

The Preferred Alternative: concatenation

As you might notice, `np.r_` can quickly become very difficult to parse. For this reason, it’s probably better to use `np.concatenate` for general concatenation, or `np.vstack`/`np.hstack`/`np.dstack` for specific cases of multi-dimensional concatenation. For example, the above expression could also be written with a few more keystrokes in a much clearer way using vertical stacking:

```
np.vstack([range(3),
          np.linspace(1, 2, 3),
          [3, 4, 3]])
array([[ 0. , 1. , 2. ],
```



```
[ 1. ,  1.5,  2. ],
 [ 3. ,  4. ,  3. ]])
```

np.c_: concatenation along columns

In case `np.r_` was not concise enough for you, you can also use `np.c_`. The expression `np.c_[*]` is simply shorthand for `np.r_['-1, 2, 0', *]`, where `*` can be replaced with any list of objects. The result is that one-dimensional arguments are stacked horizontally as columns of the two-dimensional result:

```
np.c_[:3, 1:4:3j, [1, 1, 1]]
array([[ 0. ,  1. ,  1. ],
       [ 1. ,  2.5,  1. ],
       [ 2. ,  4. ,  1. ]])
```

This is useful because stacking vectors in columns is a common operation. As with `np.r_` above, this sort of operation can be more clearly expressed using some form of concatenation or stacking, along with a transpose:

```
np.vstack([np.arange(3),
           np.linspace(1, 4, 3),
           [1, 1, 1]]).transpose()
array([[ 0. ,  1. ,  1. ],
       [ 1. ,  2.5,  1. ],
       [ 2. ,  4. ,  1. ]])
```

Why Index Tricks?

In all the cases above, we've seen that the index trick functionality is extremely concise, but this comes along with potential confusion for anyone reading the code. Throughout we've recommended avoiding these and using slightly more verbose (and much more clear) alternatives. The reason we even took the time to cover these is that they *do* sometimes appear in the wild, and it's good to know that they exist – if only so that you can properly understand them and convert them to more readable code.

Sorting Arrays

This section covers algorithms related to sorting NumPy arrays. These algorithms are a favorite topic in introductory computer science courses: if you've ever taken one, you probably have had dreams (or, depending on your temperament, nightmares) about *insertion sorts*, *selection sorts*, *merge sorts*, *quick sorts*, *bubble sorts*, and many, many more. All are means of accomplishing a similar task: sorting the values in a list or array.

For example, a simple *selection sort* repeatedly finds the minimum value from a list, and makes swaps until the list is sorted. We can code this in just a few lines of Python:

```
import numpy as np

def selection_sort(L):
    for i in range(len(L)):
        swap = i + np.argmin(L[i:])
        (L[i], L[swap]) = (L[swap], L[i])
    return L

L = np.array([2, 1, 4, 3, 5])
selection_sort(L)
array([1, 2, 3, 4, 5])
```

As any first-year Computer Science major will tell you, the selection sort is useful for its simplicity, but is much too slow to be used in practice. The reason is that as lists get big, it does not scale well. For a list of N values, it requires N loops, each of which does $\propto N$ comparisons to find the swap value. In terms of the “big-O” notation often used to characterize these algorithms, selection sort averages $\mathcal{O}[N^2]$: if you double the number of items in the list, the execution time will go up by about a factor of four.

Even selection sort, though, is much better than my all-time favorite sorting algorithms, the *bogosort*:

```
def bogosort(L):
    while np.any(L[:-1] > L[1:]):
        np.random.shuffle(L)
    return L

L = np.array([2, 1, 4, 3, 5])
bogosort(L)
array([1, 2, 3, 4, 5])
```

This silly sorting method relies on pure chance: it shuffles the array repeatedly until the result happens to be sorted. With an average scaling of $\mathcal{O}[N \times N!]$, this should (quite obviously) never be used.

Fortunately, Python contains built-in sorting algorithms which are *much* more efficient than either of the simplistic algorithms shown above. We’ll start by looking at the Python built-ins, and then take a look at the routines included in NumPy and optimized for NumPy arrays.

Sidebar: Big-O Notation

Big-O notation is a means of describing how the number of operations required for an algorithm scales as the size of the input grows. To use it correctly is to dive deeply into the realm of computer science theory, and to carefully distinguish it from the related small-o notation, big- θ notation, big- Ω notation, and probably many mutant hybrids thereof. These distinctions add precision to statements about algorithmic scaling. Outside computer science theory exams and the remarks of pedantic blog commenters, though, you’ll rarely see such distinctions made in practice. Far more

common in the data science world is a less rigid use of big-O notation: as a general (if imprecise) description of the scaling of an algorithm. With apologies to theorists and pedants, this is the interpretation we'll use throughout this book.

Big-O notation, in this loose sense, tells you how much time your algorithm will take as you increase the amount of data. If you have an $\mathcal{O}[N]$ (read “order N ”) algorithm which takes one second to operate on a list of length $N=1000$, then you should expect it to take about 5 seconds for a list of length $N=5000$. If you have an $\mathcal{O}[N^2]$ (read “order N squared”) algorithm which takes 1 second for $N=1000$, then you should expect it to take about 25 seconds for $N=5000$.

For our purposes, the N will usually indicate some aspect of the size of the data set: how many distinct objects we are looking at, how many features each object has, etc. When trying to analyze billions or trillions of samples, the difference between $\mathcal{O}[N]$ and $\mathcal{O}[N^2]$ can be far from trivial!

Notice that the big-O notation by itself tells you nothing about the actual wall-clock time of a computation, but only about its scaling as you change N . Generally, for example, an $\mathcal{O}[N]$ algorithm is considered to have better scaling than an $\mathcal{O}[N^2]$ algorithm, and for good reason. But for small datasets in particular the algorithm with better scaling might not be faster! For a particular problem, an $\mathcal{O}[N^2]$ algorithm might take 0.01sec, while a “better” $\mathcal{O}[N]$ algorithm might take 1 sec. Scale up N by a factor of 1000, though, and the $\mathcal{O}[N]$ algorithm will win out.

Even this loose version of Big-O notation can be very useful when comparing the performance of algorithms, and we'll use this notation throughout the book when talking about how algorithms scale.

Fast Sorts in Python

Python has a `list.sort` function and a `sorted` function, both of which use the *Tim-sort* algorithm (named after its creator, Tim Peters) which has average and worst-case complexity $\mathcal{O}[N \log N]$. Python's `sorted()` function will return a sorted version of any iterable without modifying the original:

```
L = [2, 1, 4, 3, 5]
sorted(L)
[1, 2, 3, 4, 5]

print(L)
[2, 1, 4, 3, 5]
```

While the `list.sort` method works sorts a list in place:

```
L.sort()
print(L)
[1, 2, 3, 4, 5]
```

There are additional arguments to each of these sorting routines which allow you to customize how items are compared; for more information, refer to the Python documentation.

Fast Sorts in NumPy: `np.sort` and `np.argsort`

Just as NumPy has a `np.sum` which is faster on arrays than Python's built-in `sum`, NumPy also has a `np.sort` which is faster on arrays than Python's built-in `sorted` function. NumPy's version uses the *quicksort* algorithm by default, though you can specify whether you'd like to use *mergesort* or *heapsort* instead. All three are $\mathcal{O}[N \log N]$, and each has advantages and disadvantages that you can read about elsewhere. For most applications, the default quicksort is more than sufficient.

To return a sorted version of the array without modifying the input, you can use `np.sort`:

```
x = np.array([2, 1, 4, 3, 5])
np.sort(x)
array([1, 2, 3, 4, 5])
```

If you prefer to sort the array in-place, you can instead use the `array.sort` method:

```
x.sort()
print(x)
[1 2 3 4 5]
```

A related function is `argsort`, which instead of returning a sorted list returns the list of indices in sorted order:

```
x = np.array([2, 1, 4, 3, 5])
i = np.argsort(x)
print(i)
[1 0 3 2 4]
```

The first element of this result gives the index of the smallest element, the second value gives the index of the second smallest, etc. These indices can then be used to construct the sorted array if desired:

```
x[i]
array([1, 2, 3, 4, 5])
```

Sorting Along Rows or Columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multi-dimensional array using the `axis` argument. With it, we can sort along rows or columns of a two-dimensional array:

```
rand = np.random.RandomState(42)
X = rand.randint(0, 10, (4, 6))
print(X)
[[6 3 7 4 6 9]
```

```

[2 6 7 4 3 7]
[7 2 5 4 1 7]
[5 1 4 0 9 5]]

# sort each column of X
np.sort(X, axis=0)
array([[2, 1, 4, 0, 1, 5],
       [5, 2, 5, 4, 3, 7],
       [6, 3, 7, 4, 6, 7],
       [7, 6, 7, 4, 9, 9]])

# sort each row of X
np.sort(X, axis=1)
array([[3, 4, 6, 6, 7, 9],
       [2, 3, 4, 6, 7, 7],
       [1, 2, 4, 5, 7, 7],
       [0, 1, 4, 5, 5, 9]])

```

Keep in mind that this treats each row or column as an independent array, and any relationships between the row or column values will be lost!

Partial Sorts: Partitioning

Sometimes we don't care about sorting the entire array, but simply care about finding the N smallest values in the array. NumPy provides this in the `np.partition` function. `np.partition` takes an array and a number K ; the result is a new array with the smallest K values to the left of the partition, and the remaining values to the right, in arbitrary order:

```

x = np.array([7, 2, 3, 1, 6, 5, 4])
np.partition(x, 3)
array([2, 1, 3, 4, 6, 5, 7])

```

Note that the first three values in the resulting array are the three smallest in the array, and the remaining array positions contain the remaining values. Within the two partitions, the elements have arbitrary order.

Similarly to sorting, we can partition along an arbitrary axis of a multi-dimensional array:

```

np.partition(X, 2, axis=1)
array([[3, 4, 6, 7, 6, 9],
       [2, 3, 4, 7, 6, 7],
       [1, 2, 4, 5, 7, 7],
       [0, 1, 4, 5, 9, 5]])

```

The result is an array where the first two slots in each row contain the smallest values from that row, with the remaining values filling the remaining slots.

Finally, just as there is a `np.argsort` which computes indices of the sort, there is a `np.argpartition` which computes indices of the partition. We'll see this in action below.

Example: K Nearest Neighbors

Let's quickly see how we might use this `argsort` function along multiple axes to find the nearest neighbors of each point in a set. We'll start by creating a random set of ten points on a two-dimensional plane. Using the standard convention, we'll arrange these in a 10×2 array:

```
X = rand.rand(10, 2)
```

To get an idea of how these points look, let's quickly scatter-plot them, using tools explored in chapter X.X:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # Plot styling
plt.scatter(X[:, 0], X[:, 1], s=100);
```

Now we'll compute the distance between each pair of points. Recall that the distance $D_{x,y}$ between a point x and a point y in d dimensions satisfies

$$\|D\{x,y\}^2 = \sum_{i=1}^d (x_i - y_i)^2$$

We can keep in mind that sorting according to D^2 is equivalent to sorting according to D . Using the broadcasting rules covered in section X.X along with the aggregation routines from section X.X, we can compute the matrix of distances in a single line of code:

```
dist_sq = np.sum((X[:, np.newaxis, :] - X[np.newaxis, :, :]) ** 2, axis=-1)
```

The above operation has a lot packed into it, and it might be a bit confusing if you're unfamiliar with NumPy's broadcasting rules. When you come across code like this, it can be useful to mentally break it down into steps:

```
# for each pair of points, compute differences in their coordinates
differences = X[:, np.newaxis, :] - X[np.newaxis, :, :]
differences.shape
(10, 10, 2)

# square the coordinate differences
sq_differences = differences ** 2
sq_differences.shape
(10, 10, 2)

# sum the coordinate differences to get the squared distance
dist_sq = sq_differences.sum(-1)
dist_sq.shape
(10, 10)
```

Just to double-check what we are doing, we should see that the diagonal of this matrix (i.e. the set of distances between each point and itself) is all zero:

```
dist_sq.diagonal()
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

It checks out! With the pairwise square-distances converted, we can now use `np.argsort` to sort along each row. The left-most columns will then give the indices of the nearest neighbors:

```
nearest = np.argsort(dist_sq, axis=1)
print(nearest)
[[0 3 9 7 1 4 2 5 6 8]
 [1 4 7 9 3 6 8 5 0 2]
 [2 1 4 6 3 0 8 9 7 5]
 [3 9 7 0 1 4 5 8 6 2]
 [4 1 8 5 6 7 9 3 0 2]
 [5 8 6 4 1 7 9 3 2 0]
 [6 8 5 4 1 7 9 3 2 0]
 [7 9 3 1 4 0 5 8 6 2]
 [8 5 6 4 1 7 9 3 2 0]
 [9 7 3 0 1 4 5 8 6 2]]
```

Notice that the first column gives the numbers zero through nine in order: this is due to the fact that each point's closest neighbor is itself! But by using a full sort, we've actually done more work than we need to in this case. If we're simply interested in the nearest *K* neighbors, all we need is to partition each row so that the smallest three squared distances come first, with larger distances filling the remaining positions of the array. We can do this with the `np.argpartition` function:

```
K = 2
nearest_partition = np.argpartition(dist_sq, K + 1, axis=1)
```

In order to visualize this network of neighbors, let's quickly plot the points along with lines representing the connections from each point to its two nearest neighbors:

```
plt.scatter(X[:, 0], X[:, 1], s=100)

# draw lines from each point to its K nearest neighbors
K = 2

for i in range(X.shape[0]):
    for j in nearest_partition[i, :K+1]:
        # plot a line from X[i] to X[j]
        # use some zip magic to make it happen:
        plt.plot(*zip(X[j], X[i]), color='black')
```

Each point in the plot has lines drawn to its two nearest neighbors. At first glance, it might seem strange that some of the points have more than two lines coming out of them: this is due to the fact that if point A is one of the two nearest neighbors of point B, this does not necessarily imply that point B is one of the two nearest neighbors of point A.

Though the broadcasting and row-wise sorting of the above approach might seem less straightforward than writing a loop, it turns out to be a very efficient way of operating on this data in Python. You might be tempted to do the same type of operation

by manually looping through the data and sorting each set of neighbors individually, but this would almost certainly lead to a slower algorithm than the vectorized version we used above. The beauty of the above approach is that it's written in a way that's agnostic to the size of the input data: we could just as easily compute the neighbors among 100 or 1,000,000 points in any number of dimensions, and the code would look the same.

Finally, we should note that when doing very large nearest neighbor searches, there are tree-based algorithms or approximate algorithms that can scale as $\mathcal{O}[N \log N]$ or better rather than the $\mathcal{O}[N^2]$ of the brute-force algorithm above. For more information on these fast tree-based K-neighbors queries, see section X.X.

Searching and Counting Values In Arrays

This section covers ways of finding a particular value or particular values in an array of data. This may sound like it simply requires a linear scan of the data, but when you know the data is sorted, or if you're trying to find multiple values at the same time, there are faster ways to go about it.

Python Standard Library Tools

Because this is so important, the Python standard library has a couple solutions you should be aware of. Here we'll quickly go over the functions and methods Python contains for doing searches in unsorted and sorted lists, before continuing on to the more specialized tools in NumPy.

Unsorted Lists

For any arbitrary list of data, the only way to find whether an item is in the list is to scan through it. Rather than making you write this loop yourself, Python provides the `list.index` method, which scans the list looking for the first occurrence of a value, and returning its index:

```
L = [5, 2, 6, 1, 3, 6]
L.index(6)
2
```

Sorted Lists

If your list is sorted, there is a more sophisticated approach based on recursively bisecting the list, and narrowing-in on the half which contains the desired value. Python implements this $\mathcal{O}[N \log N]$ in the built-in `bisect` package:

```
import bisect
L = [2, 4, 5, 5, 7, 9]
bisect.bisect_left(L, 7)
4
```


Technically, `bisect` is not searching for the value itself, but for the *insertion index*: that is, the index at which the value should be inserted in order to keep the list sorted:

```
bisect.bisect(L, 4.5)
2

L.insert(2, 4.5)
L == sorted(L)
True
```

If your goal is to insert items into the list, the `bisect.insort` function is a bit more efficient. For more details on `bisect` and tools therein, use IPython's help features or refer to Python's online documentation.

Searching for Values in NumPy Arrays

NumPy has some similar tools and patterns which work for quickly locating values in arrays, but their use is a bit different than the Python standard library approaches. The patterns listed below have the benefit of operating much more quickly on NumPy arrays, and of scaling well as the size of the array grows.

Unsorted Arrays

For finding values in unsorted data, NumPy has no strict equivalent of `list.index`. Instead, it is typical to use a pattern based on masking and the `np.where` function (see section X.X)

```
import numpy as np
A = np.array([5, 2, 6, 1, 3, 6])

np.where(A == 6)
(array([2, 5]),)
```

`np.where` always returns a *tuple* of index arrays; even in the case of a one-dimensional array you should remember that the output is a one-dimensional tuple. To isolate the first index at which the value is found, then, you must use `[0]` to access the first item of the tuple, and `[0]` again to access the first item of the array of indices. This gives the equivalent result to `list.index`:

```
list(A).index(6) == np.where(A == 6)[0][0]
True
```

Note that this masking approach solves a different use-case than `list.index`: rather than finding the first occurrence of the value and stopping, it finds all occurrences of the value simultaneously. If `np.where` is a bottleneck and your code requires quickly finding the first occurrence of a value in an unsorted list, it will require writing a simple utility with Cython, Numba, or a similar tool; see chapter X.X.

Sorted Arrays

If your array is sorted, NumPy provides a fast equivalent of Python's `bisect` utilities with the `np.searchsorted` function:

```
A = np.array([2, 4, 5, 5, 7, 9])
np.searchsorted(A, 7)
4
```

Like `bisect`, `np.searchsorted` returns an *insertion index* for the given value:

```
np.searchsorted(A, 4.5)
2
```

Unlike `bisect`, it can search for insertion indices for multiple values in one call, without the need for an explicit loop:

```
np.searchsorted(A, [7, 4.5, 5, 10, 0])
array([4, 2, 2, 6, 0])
```

The `searchsorted` function has a few other options, which you can read about in the functions docstring.

Counting and Binning

A related set of functionality in NumPy is the built-in tools for counting and binning of values. For example, to count the occurrences of a value or other condition in an array, use `np.count_nonzero`:

```
A = np.array([2, 0, 2, 3, 4, 3, 4, 3])
np.count_nonzero(A == 4)
2
```

`np.unique` for counting

For counting occurrences of all values at once, you can use the `np.unique` function, which in NumPy versions 1.9 or later has a `return_count` option:

```
np.unique(A, return_counts=True)
(array([0, 2, 3, 4]), array([1, 2, 3, 2]))
```

The first return value is the list of unique values in the array, and the second return value is the list of associated counts for those values.

`np.bincount`. If your data consist of positive integers, a more compact way to get this information is with the `np.bincount` function:

```
np.bincount(A)
array([1, 0, 2, 3, 2])
```

If `counts` is the output of `np.bincount(A)` then `counts[val]` is the number of times value `val` occurs in the array `A`.

np.histogram. `np.bincount` can become cumbersome when your values are large, and it does not apply when your values are not integers. For this more general case, you can specify bins for your values with the `np.histogram` function:

```
counts, bins = np.histogram(A, bins=range(6))
print("bins:      ", bins)
print("counts in bin: ", counts)
bins:          [0 1 2 3 4 5]
counts in bin: [1 0 2 3 2]
```

Here the output is formatted to make clear that the bins give the *boundaries* of the range, and the counts indicate how many values fall within each of those ranges. For this reason, the counts array will have one fewer entry than the bins array. A related function to be aware of is `np.digitize`, which quickly computes index of the appropriate bin for a series of values.

Structured Data: NumPy's Structured Arrays

While often our data can be well-represented by a homogeneous array of values, sometimes this is not the case. This section demonstrates the use of NumPy's *structured arrays* and *record arrays*, which provide efficient storage for compound, heterogeneous data. While the patterns shown here are useful for simple operations, scenarios like this often lend themselves to the use of Pandas Dataframes, which we'll explore in the next chapter.

Imagine that we have several categories of data on a number of people (say, name, age, and weight), and we'd like to store these values for use in a Python program. It would be possible to store these in three separate arrays:

```
name = ['Alice', 'Bob', 'Cathy', 'Doug']
age = [25, 45, 37, 19]
weight = [55.0, 85.5, 68.0, 61.5]
```

But this is a bit clumsy. There's nothing here that tells us that the three arrays are related; it would be more natural if we could use a single structure to store all of this data. NumPy can handle this through structured arrays, which are arrays with compound data types.

Recall that previously we created a simple array using an expression like this:

```
x = np.zeros(4, dtype=int)
```

We can similarly create a structured array using a compound data type specification:

```
# Use a compound data type for structured arrays
data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
```

```

        'formats':('U10', 'i4', 'f8'))}

print(data.dtype)
[('name', '<U10'), ('age', '<i4'), ('weight', '<f8')]

```

Here 'U10' translates to “unicode string of maximum length 10”, 'i4' translates to “4-byte (i.e. 32 bit) integer” and 'f8' translates to “8-byte (i.e. 64 bit) float”. We’ll discuss other options for these type codes below.

Now that we’ve created an empty container array, we can fill the array with our lists of values:

```

data['name'] = name
data['age'] = age
data['weight'] = weight
print(data)
[('Alice', 25, 55.0) ('Bob', 45, 85.5) ('Cathy', 37, 68.0)
 ('Doug', 19, 61.5)]

```

As we had hoped, the data is now arranged together in one convenient block of memory.

The handy thing with structured arrays is that you can now refer to values either by index or by name:

```

# Get all names
data['name']
array(['Alice', 'Bob', 'Cathy', 'Doug'],
      dtype='<U10')

# Get first row of data
data[0]
('Alice', 25, 55.0)

# Get the name from the last row
data[-1]['name']
'Doug'

```

Using boolean masking, this even allows you to do some more sophisticated operations such as filtering on age:

```

# Get names where age is under 30
data[data['age'] < 30]['name']
array(['Alice', 'Doug'],
      dtype='<U10')

```

Note that if you’d like to do any operations which are much more complicated than these, you should probably consider the Pandas package, covered in the next section. Pandas provides a Dataframe object, which is a structure built on NumPy arrays that offers a variety of useful data manipulation functionality similar to what we’ve shown above, as well as much, much more.

Creating Structured Arrays

Structured array data types can be specified in a number of ways. Above, we saw the dictionary method:

```
np.dtype({'names':('name', 'age', 'weight'),
         'formats':('U10', 'i4', 'f8')})
dtype([('name', '<U10'), ('age', '<i4'), ('weight', '<f8')])
```

For clarity, numerical types can be specified using Python types or NumPy dtypes instead:

```
np.dtype({'names':('name', 'age', 'weight'),
         'formats':((np.str_, 10), int, np.float32)})
dtype([('name', '<U10'), ('age', '<i8'), ('weight', '<f4')])
```

A compound type can also be specified as a list of tuples:

```
np.dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])
dtype([('name', 'S10'), ('age', '<i4'), ('weight', '<f8')])
```

If the names of the types do not matter to you, you can specify the types alone in a comma-separated string:

```
np.dtype('S10,i4,f8')
dtype([('f0', 'S10'), ('f1', '<i4'), ('f2', '<f8')])
```

The shortened string format codes may seem confusing, but they are built on simple principles. The first (optional) character is < or >, which means “little endian” or “big endian” respectively, and specifies the ordering convention for significant bits. The next character specifies the type of data: characters, bytes, ints, floating points, etc. (see the table below). The last character or characters represents the size of the object in bytes.

character	description	example
'b'	byte	<code>np.dtype('b')</code>
'i'	(signed) integer	<code>np.dtype('i4') == np.int32</code>
'u'	unsigned integer	<code>np.dtype('u1') == np.uint8</code>
'f'	floating point	<code>np.dtype('f8') == np.int64</code>
'c'	complex floating point	<code>np.dtype('c16') == np.complex128</code>
'S', 'a'	string	<code>np.dtype('S5')</code>
'U'	unicode string	<code>np.dtype('U') == np.str_</code>
'V'	raw data (void)	<code>np.dtype('V') == np.void</code>

More Advanced Compound Types

It is possible to define even more advanced compound types. For example, you can create a type where each element contains an array or matrix of values. Here, we'll create a data type with a `mat` component consisting of a 3x3 floating point matrix:

```
tp = np.dtype([('id', 'i8'), ('mat', 'f8', (3, 3))])
X = np.zeros(1, dtype=tp)
print(X[0])
print(X['mat'][0])
(0, [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]])
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]
```

Now each element in the `X` array consists of an `id` and a 3x3 matrix. Why would you use this rather than a simple multi-dimensional array, or perhaps a Python dictionary? The reason is that this numpy dtype directly maps onto a C structure definition, so the buffer containing the array content can be accessed directly within an appropriately written C program. We'll see examples of this type of pattern in section X.X, when we discuss Cython, a C-enabled extension of the Python language.

RecordArrays: Structured Arrays with a Twist

NumPy also provides the `np.recarray` class, which is almost identical to the structured arrays described above with one additional feature: fields can be accessed as attributes rather than as dictionary keys. Recall that we previously accessed the ages by writing:

```
data['age']
array([25, 45, 37, 19], dtype=int32)
```

If we view our data as a record array instead, we can access this with slightly fewer keystrokes:

```
data_rec = data.view(np.recarray)
data_rec.age
array([25, 45, 37, 19], dtype=int32)
```

The downside is that for record arrays, there is some extra overhead involved in accessing the fields, even when using the same syntax. We can see this here:

```
%timeit data['age']
%timeit data_rec['age']
%timeit data_rec.age
1000000 loops, best of 3: 241 ns per loop
100000 loops, best of 3: 4.61 µs per loop
100000 loops, best of 3: 7.27 µs per loop
```

Whether the more convenient notation is worth the additional overhead will depend on your own application.

On to Pandas

This section on structured and record arrays is purposely at the end of the NumPy section, because it leads so well into the next chapter: Pandas. Structured arrays like the ones above are good to know about for certain situations, especially in case you're using NumPy arrays to map onto binary data formats in C, Fortran, or another language. For day-to-day use of structured data, the Pandas package is a much better choice. We'll take a look at that package next.

Introduction to Pandas

In the last section we dove into detail on NumPy and its `ndarray` object, which provides efficient storage and manipulation of dense typed arrays in Python. Here we'll build on this knowledge by looking in detail at the data structures provided by the Pandas library. Pandas is a newer package built on top of NumPy, and provides an efficient implementation of a *Data Frame*. Data frames are essentially multi-dimensional arrays with attached row and column labels, and often with heterogeneous types and/or missing data. As well as offering a convenient storage interface for labeled data, Pandas implements a number of powerful data operations familiar to users of both database frameworks and spreadsheet programs.

As we saw, NumPy's `ndarray` data structure provides essential features for the type of clean, well-organized data typically seen in numerical computing tasks. While it serves this purpose very well, its limitations become clear when we need more flexibility (such as attaching labels to data, working with missing data, etc.) and when attempting operations which do not map well to element-wise broadcasting (such as groupings, pivots, etc.), each of which is an important piece of analyzing the less structured data available in many forms in the world around us. Pandas, and in particular its `Series` and `DataFrame` objects, builds on the NumPy array structure and provides efficient access to these sorts of “data munging” tasks that occupy most of a data scientist's time.

In this chapter, we will focus on the mechanics of using the `Series`, `DataFrame`, and related structures effectively. We will use examples drawn from real datasets where appropriate, but these examples are not necessarily the focus. Like the previous chapter on NumPy, the primary purpose of this chapter is to serve as a comprehensive introduction to the Python tools that will enable the more in-depth analyses and discussions of the second half of the book.

Installing and Using Pandas

Installation of Pandas on your system requires a previous install of NumPy, as well as the appropriate tools to compile the C and Cython sources on which Pandas is built. Details on this installation can be found in the Pandas documentation: <http://pandas.pydata.org/>. If you followed the advice in the introduction and used the Anaconda stack, you will already have Pandas installed.

Once Pandas is installed, you can import it and check the version:

```
import pandas
pandas.__version__
'0.16.1'
```

Just as we generally import NumPy under the alias `np`, we will generally import Pandas under the alias `pd`:

```
import pandas as pd
```

This import convention will be used throughout the remainder of this book.

Reminder about Built-in Documentation

As you read through this chapter, don't forget that IPython gives you the ability to quickly explore the contents of a package (by using the tab-completion feature), as well as the documentation of various functions (using the `"?"` character).

For example, you can type

```
In [3]: pd.<TAB>
```

to display all the contents of the `pandas` namespace, and

```
In [4]: pd?
```

to display Pandas's built-in documentation. More detailed documentation, along with tutorials and other resources, can be found at <http://pandas.pydata.org/>.

Introducing Pandas Objects

At the very basic level, Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices. As we will see through the rest of this chapter, Pandas provides a host of useful tools, methods, and functionality on top of these data structures, but nearly everything that follows will require an understanding of what these structures are. This first section will cover the three fundamental Pandas data structures: the `Series`, `DataFrame`, and `Index`.

Just as the standard alias for importing numpy is np, the standard alias for importing pandas is pd:

```
import numpy as np
import pandas as pd
```

Pandas Series

A pandas Series is a one-dimensional array of indexed data. It can be created from a list or array as follows:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

As we see in the output above, the series has both a sequence of values and a sequence of indices, which we can access with the `values` and `index` attributes. The `values` are simply a familiar NumPy array:

```
data.values
array([ 0.25,  0.5 ,  0.75,  1.  ])
```

while the `index` is an array-like object of type `pd.Index`, which we'll discuss in more detail below.

```
data.index
Int64Index([0, 1, 2, 3], dtype='int64')
```

Like with a NumPy array, data can be accessed by the associated index via the familiar Python square-bracket notation:

```
data[1]
0.5

data[1:3]
1    0.50
2    0.75
dtype: float64
```

As we will see, though, the Pandas series is much more general and flexible than the one-dimensional NumPy array that it emulates.

Series as Generalized NumPy Array

From what we've seen so far, it may look like the `Series` object is basically interchangeable with a one-dimensional NumPy array. The essential difference is the presence of the index: while the NumPy Array has an *implicitly defined* integer index used

to access the values, the Pandas Series has an *explicitly defined* index associated with the values.

This explicit index definition gives the Series object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type. For example, if we wish, we can use strings as an index:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                  index=['a', 'b', 'c', 'd'])

data
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

and the item access works as expected:

```
data['b']
0.5
```

We can even use non-contiguous or non-sequential indices

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                  index=[2, 5, 3, 7])

data
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64

data[5]
0.5
```

Series as Specialized Dictionary

In this way, you can think of a Pandas Series a bit like a specialization of a Python dictionary. A dictionary is a structure which maps arbitrary keys to a set of arbitrary values, and a series is a structure which maps *typed* keys to a set of *typed* values. This typing is important: just as the type-specific compiled code behind a NumPy array makes it more efficient than a Python list for certain operations, the type information of a Pandas Series makes it much more efficient than Python dictionaries for certain operations.

The series-as-dict analogy can be made even more clear by constructing a Series object directly from a Python dictionary:

```
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
```

```

        'Illinois': 12882135}
population = pd.Series(population_dict)
population
California    38332521
Florida       19552860
Illinois      12882135
New York      19651127
Texas         26448193
dtype: int64

```

By default, a series will be created where the index is drawn from the sorted keys. From here, typical dictionary-style item access can be performed:

```

population['California']
38332521

```

Unlike a dictionary, though, the Series also supports array-style operations such as slicing:

```

population['California':'Illinois']
California    38332521
Florida       19552860
Illinois      12882135
dtype: int64

```

We'll discuss some of the quirks of Pandas indexing and slicing in Section X.X.

Constructing Series Objects

We've already seen a few ways of constructing a Pandas Series from scratch; all of them are some version of the following,

```
>>> pd.Series(data, index=index)
```

where `index` is an optional argument, and `data` can be one of many entities.

For example, `data` can be a list or NumPy array, in which case `index` defaults to an integer sequence:

```

pd.Series([2, 4, 6])
0    2
1    4
2    6
dtype: int64

```

`data` can be a scalar, which is broadcast to fill the specified index:

```

pd.Series(5, index=[100, 200, 300])
100    5
200    5
300    5
dtype: int64

```

`data` can be a dictionary, in which `index` defaults to the sorted dictionary keys:

```
pd.Series({2:'a', 1:'b', 3:'c'})
1    b
2    a
3    c
dtype: object
```

In each case, the index can be explicitly set if a different result is preferred:

```
pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])
3    c
2    a
dtype: object
```

Notice that here we explicitly identified the particular indices to be included from the dictionary.

Pandas DataFrame

The next fundamental structure in Pandas is the `DataFrame`. Like the `Series` above, the `DataFrame` can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary. We'll discuss these views below.

DataFrame as a Generalized NumPy array

If a `Series` is an analog of a one-dimensional array with flexible indices, a `DataFrame` is an analog of a two-dimensional array with both flexible row indices and flexible column names. Just as you might think of a two-dimensional array as an ordered sequence of aligned one-dimensional columns, you can think of a `DataFrame` as a sequence of aligned `Series` objects. Here, by “aligned” we mean that they share the same index.

To demonstrate this, let's first construct a new `Series` listing the area of each of the five states mentioned above:

```
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297, 'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
area
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
dtype: int64
```

Now that we have this along with the population `Series` from above, we can use a dictionary to construct a single two-dimensional object containing this information:

```
states = pd.DataFrame({'population': population,
                       'area': area})
states
```

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

Like the Series object, the DataFrame has an `index` attribute which gives access to the index labels:

```
states.index
Index(['California', 'Florida', 'Illinois', 'New York', 'Texas'], dtype='object')
```

Additionally, the DataFrame has a `columns` attribute which is an Index object holding the column labels:

```
states.columns
Index(['area', 'population'], dtype='object')
```

Thus the DataFrame can be thought of as a generalization of a two-dimensional NumPy array, where both the rows and columns have a generalized index for accessing the data.

DataFrame as Specialized Dictionary

Similarly, we can think of a dataframe as a specialization of a dictionary. Where a dictionary maps a key to a value, a data frame maps a column name to a Series of column data. For example, asking for the 'area' attribute returns the Series object containing the areas we saw above:

```
states['area']
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
Name: area, dtype: int64
```

Notice the potential point of confusion here: in a two-dimensional NumPy array, `data[0]` will return the first *row*. For a dataframe, `data['col0']` will return the first *column*. Because of this, it is probably better to think about dataframes as generalized dictionaries rather than generalized arrays, though both ways of looking at the situation can be useful. We'll explore more flexible means of indexing DataFrames in Section X.X.

Constructing DataFrame Objects

A Pandas DataFrame can be constructed in a variety of ways. Here we'll give several examples:

From a single Series object. A DataFrame is a collection of series, and a single-column dataframe can be constructed from a single series:

```
pd.DataFrame(population, columns=['population'])
```

	population
California	38332521
Florida	19552860
Illinois	12882135
New York	19651127
Texas	26448193

From a list of dicts. Any list of dictionaries can be made into a dataframe. We'll use a simple list comprehension to create some data:

```
data = [{'a': i, 'b': 2 * i}
        for i in range(3)]
pd.DataFrame(data)
```

	a	b
0	0	0
1	1	2
2	2	4

Even if some keys in the dictionary are missing, Pandas will fill them in with NaN (i.e. “not a number”) values:

```
pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

	a	b	c
0	1	2	NaN
1	NaN	3	4

From a dictionary of Series objects. We saw this above, but a DataFrame can be constructed from a dictionary of Series objects works as well:

```
pd.DataFrame({'population': population,
              'area': area})
```

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

From a two-dimensional NumPy array. Given a two-dimensional array of data, we can create a dataframe with any specified column and index names. If left out, an integer index will be used for each.

```
pd.DataFrame(np.random.rand(3, 2),
              columns=['foo', 'bar'],
              index=['a', 'b', 'c'])
```

	foo	bar
a	0.201512	0.332724

```
b 0.905656 0.281871
c 0.455450 0.530707
```

From a numpy structured array. We covered structured arrays in section X.X. A Pandas dataframe operates much like a structured array, and can be created directly from one:

```
A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
A
array([(0, 0.0), (0, 0.0), (0, 0.0)],
      dtype=[('A', '<i8'), ('B', '<f8')])

pd.DataFrame(A)
   A  B
0  0  0
1  0  0
2  0  0
```

Pandas Index

Above we saw that both the `Series` and `DataFrame` contain an explicit *index* which lets you reference and modify data. This `Index` object is an interesting structure in itself, and it can be thought of either as an *immutable array* or as an *ordered set*. Those views have some interesting consequences in the operations available on `Index` objects. As a simple example, let's construct an index from a list of integers:

```
ind = pd.Index([2, 3, 5, 7, 11])
ind
Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

Index as Immutable Array

The index in many ways operates like an array. For example, we can use standard Python indexing notation to retrieve values or slices:

```
ind[1]
3

ind[:2]
Int64Index([2, 5, 11], dtype='int64')
```

`Index` objects also have many of the attributes familiar from NumPy arrays:

```
print(ind.size, ind.shape, ind.ndim, ind.dtype)
5 (5,) 1 int64
```

One difference between `Index` objects and NumPy arrays is that indices are immutable: that is, they cannot be modified via the normal means:

```
ind[1] = 0
```


This immutability makes it safer to share indices between multiple dataframes and arrays, without the potential for nasty side-effects from inadvertent index modification.

Index as Ordered Set

Pandas objects are designed to facilitate operations such as joins across datasets, which depend on many aspects of set arithmetic. Recall that Python has a built-in `set` object, which we explored in section X.X. The `Index` object follows many of the conventions of this built-in `set` object, so that unions, intersections, differences, and other combinations can be computed in a familiar way:

```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])

indA & indB # intersection
Int64Index([3, 5, 7], dtype='int64')

indA | indB # union
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')

indA - indB # difference
Int64Index([-1, 0, 0, 0, -2], dtype='int64')

indA ^ indB # symmetric difference
Int64Index([1, 2, 9, 11], dtype='int64')
```

These operations may also be accessed via object methods, e.g. `indA.intersection(indB)`. For more information on the variety of set operations implemented in Python, see section X.X. Nearly every syntax listed there, with the exception of operations which modify the set, can also be performed on `Index` objects.

Looking Forward

Above we saw the basics of the `Series`, `DataFrame`, and `Index` objects, which form the foundation of data-oriented computing with Pandas. We saw how they are similar to and different from other Python data structures, and how they can be created from scratch from these more familiar objects. Through this chapter, we'll go more into more detail about creation of these structures (including very useful interfaces for creating them from various file types) and manipulating data within these structures. Just as understanding the effective use of NumPy arrays is fundamental to effective numerical computing in Python, understanding the effective use of Pandas structures is fundamental to the data munging required for data science in Python.

Data Indexing and Selection

In the previous chapter, we looked in detail at methods and tools to access, set, and modify values in NumPy arrays. These included indexing (e.g. `arr[2, 1]`), slicing

(e.g. `arr[:, 1:5]`), masking (e.g. `arr[arr > 0]`), fancy indexing (e.g. `arr[0, [1, 5]]`), and combinations thereof (e.g. `arr[:, [1, 5]]`). Here we'll look at similar means of accessing and modifying values in Pandas `Series` and `DataFrame` objects. If you have used the NumPy patterns mentioned above, the corresponding patterns in Pandas will feel very familiar, though there are a few quirks to be aware of.

We'll start with the simple case of the one-dimensional `Series` object, and then move on to the more complicated two-dimensional `DataFrame` object.

Data Selection in Series

As we saw in the previous section, a `Series` object acts in many ways like a one-dimensional NumPy array, and in many ways like a standard Python dictionary. If we keep these two overlapping analogies in mind, it will help us to understand the patterns of data indexing and selection in these arrays.

Series as dictionary

Like a dictionary, the `Series` object provides a mapping from a collection of keys to a collection of values:

```
import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])

data['b']
0.5
```

We can also use dictionary-like Python expressions and methods to examine the keys/indices and values:

```
'a' in data
True

data.keys()
Index(['a', 'b', 'c', 'd'], dtype='object')

list(data.items())
[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

`DataFrame` objects can even be modified with a dictionary-like syntax. Just as you can extend a dictionary by assigning to a new key, you can extend a series by assigning to a new index value:

```
data['e'] = 1.25
data
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64
```

This easy mutability of the objects is a convenient feature: under the hood, Pandas is making decisions about memory layout and data copying that might need to take place; the user generally does not need to worry about these issues.

Series as 1D Array

A `Series` builds on this dictionary-like interface and provides array-style item selection via *slices*, *masking*, and *fancy indexing*, examples of which can be seen below:

```
# slicing by explicit index
data['a':'c']
a    0.25
b    0.50
c    0.75
dtype: float64

# slicing by implicit integer index
data[0:2]
a    0.25
b    0.50
dtype: float64

# masking
data[(data > 0.3) & (data < 0.8)]
b    0.50
c    0.75
dtype: float64

# fancy indexing
data[['a', 'e']]
a    0.25
e    1.25
dtype: float64
```

Among these, slicing may be the source of the most confusion. Notice that when slicing with an explicit index (i.e. `data['a':'c']`), the final index is *included* in the slice, while when slicing with an implicit index (i.e. `data[0:2]`), the final index is *excluded* from the slice.

Indexers: `loc`, `iloc`, and `ix`

The slicing and indexing conventions above can be a source of confusion. For example, if your series has an explicit integer index, an indexing operation such as `data[1]` will use the explicit indices, while a slicing operation like `data[1:3]` will use the implicit Python-style index.

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
1    a
3    b
```

```

5    c
dtype: object

# explicit index when indexing
data[1]
'a'

# implicit index when slicing
data[1:3]
3    b
5    c
dtype: object

```

Because of this potential confusion in the case of integer indexes, Pandas provides some special *indexer* attributes which explicitly access certain indexing schemes. These are not functional methods, but attributes which expose a particular slicing interface to the data in the Series.

First, the `loc` attribute allows indexing and slicing which always references the explicit index:

```

data.loc[1]
'a'

data.loc[1:3]
1    a
3    b
dtype: object

```

The `iloc` attribute allows indexing and slicing which always references the implicit Python-style index:

```

data.iloc[1]
'b'

data.iloc[1:3]
3    b
5    c
dtype: object

```

A third indexing attribute, `ix`, is a hybrid of the two, and for Series objects is equivalent to standard `[]`-based indexing. The purpose of the `ix` indexer will become more apparent in the context of DataFrame objects, below.

One guiding principle of Python code (see the Zen of Python, section X.X) is that “explicit is better than implicit”. The explicit nature of `loc` and `iloc` make them very useful in maintaining clean and readable code; especially in the case of integer indexes, I recommend using these both to make code easier to read and understand, and to prevent subtle bugs due to the mixed indexing/slicing convention.

Data Selection in DataFrame

Recall that a DataFrame acts in many ways like a two-dimensional or structured array, and acts in many ways like a dictionary of Series structures sharing the same index. These analogies can be helpful to keep in mind as we explore data selection within this structure.

DataFrame as a Dictionary

The first analogy we will consider is the DataFrame as a dictionary of related Series objects. Let's return to our example of areas and populations of states:

```
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
data
```

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

The individual Series which make up the columns of the dataframe can be accessed via dictionary-style indexing of the column name:

```
data['area']
```

California	423967
Florida	170312
Illinois	149995
New York	141297
Texas	695662

Name: area, dtype: int64

Equivalently, we can use attribute-style access with column names which are strings:

```
data.area
```

California	423967
Florida	170312
Illinois	149995
New York	141297
Texas	695662

Name: area, dtype: int64

This attribute-style column access actually accesses the exact same object as the dictionary-style access:

```
data.area is data['area']
True
```

Though this is a useful shorthand, keep in mind that it does not work for all cases! For example, if the column names are not strings, or if the column names conflict with methods of the dataframe, this attribute-style access is not possible. For example, the DataFrame has a `pop` method, so `data.pop` will point to this rather than the "pop" column:

```
data.pop is data['pop']
False
```

Like with the Series objects above, this dictionary-style syntax can also be used to modify the object, in this case adding a new column:

```
data['density'] = data['pop'] / data['area']
data
```

	area	pop	density
California	423967	38332521	90.413926
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763
New York	141297	19651127	139.076746
Texas	695662	26448193	38.018740

This shows a preview of the straightforward syntax of element-by-element arithmetic between Series objects; we'll dig into this further in section X.X.

DataFrame as Two-dimensional Array

As mentioned, we can also view the dataframe as an enhanced two-dimensional array. We can examine the raw underlying data array using the `values` attribute:

```
data.values
array([[ 4.239670000e+05,  3.83325210e+07,  9.04139261e+01],
       [ 1.703120000e+05,  1.95528600e+07,  1.14806121e+02],
       [ 1.499950000e+05,  1.28821350e+07,  8.58837628e+01],
       [ 1.412970000e+05,  1.96511270e+07,  1.39076746e+02],
       [ 6.956620000e+05,  2.64481930e+07,  3.80187404e+01]])
```

With this picture in mind, many familiar array-like observations can be done on the dataframe itself. For example, we can transpose the full dataframe to swap rows and columns:

```
data.transpose()
```

	California	Florida	Illinois	New York	\
area	423967.000000	170312.000000	149995.000000	141297.000000	
pop	38332521.000000	19552860.000000	12882135.000000	19651127.000000	
density	90.413926	114.806121	85.883763	139.076746	
	Texas				
area	695662.000000				

```
pop      26448193.00000
density   38.01874
```

When it comes to indexing of `DataFrame` objects, however, it is clear that the dictionary-style indexing of columns precludes our ability to simply treat it as a NumPy array. In particular, passing a single index to an array accesses a row:

```
data.values[0]
array([ 4.23967000e+05,  3.83325210e+07,  9.04139261e+01])
```

While passing a single “index” to a dataframe accesses a column:

```
data['area']
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
Name: area, dtype: int64
```

Thus for array-style indexing, we need another convention. Here Pandas again uses the `loc`, `iloc`, and `ix` indexers mentioned above. Using the `iloc` indexer, we can index the underlying array as if it is a simple NumPy array (using the implicit Python-style index), but the `DataFrame` index and column labels are maintained in the result:

```
data.iloc[:3, :2]
      area      pop
California  423967  38332521
Florida     170312  19552860
Illinois    149995  12882135
```

Similarly, using the `loc` indexer we can index the underlying data in an array-like style but using the explicit index and column names:

```
data.loc[:'Illinois', : 'pop']
      area      pop
California  423967  38332521
Florida     170312  19552860
Illinois    149995  12882135
```

The `ix` indexer allows a hybrid of these two approaches:

```
data.ix[:3, : 'pop']
      area      pop
California  423967  38332521
Florida     170312  19552860
Illinois    149995  12882135
```

Keep in mind that for integer indices, the `ix` indexer is subject to the same potential sources of confusion as discussed for integer-indexed `Series` objects above.

Any of the familiar NumPy-style data access patterns can be used within these indexers. For example, in the `loc` indexer we can combine masking and fancy indexing as in the following:

```
data.loc[data.density > 100, ['pop', 'density']]
```

	pop	density
Florida	19552860	114.806121
New York	19651127	139.076746

Keep in mind also that any of these indexing conventions may also be used to set or modify values; this is done in the standard way that you might be used to from NumPy:

```
data.iloc[0, 2] = 90
```

```
data
```

	area	pop	density
California	423967	38332521	90.000000
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763
New York	141297	19651127	139.076746
Texas	695662	26448193	38.018740

To built-up your fluency in Pandas data manipulation, I suggest spending some time with a simple `DataFrame` and exploring the types of indexing, slicing, masking, and fancy indexing that are allowed by these various indexing approaches.

Additional Indexing Conventions

There are a couple extra indexing conventions which might seem a bit inconsistent with the above discussion, but nevertheless can be very useful in practice. First, while direct integer *indices* are not allowed on `DataFrames`, direct integer *slices* are allowed, and are taken on rows rather than on columns as you might expect:

```
data[1:3]
```

	area	pop	density
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

Similarly, direct masking operations are also interpreted row-wise rather than column-wise:

```
data[data.density > 100]
```

	area	pop	density
Florida	170312	19552860	114.806121
New York	141297	19651127	139.076746

These two conventions are syntactically similar to those on a NumPy array, and while these may not precisely fit the mold of the above conventions they are nevertheless quite useful in practice.

Summary

Here we have discussed the various ways to access and modify values within the basic Pandas data structures. With this, we're slowly building-up our fluency with manipulating and operating on labeled data within Pandas. In the next section, we'll take this a bit farther and begin to examine the types of *operations* that you can do on Pandas Series and DataFrame objects.

Operations in Pandas

One of the essential pieces of NumPy is the ability to perform quick elementwise operations, both with basic arithmetic (addition, subtraction, multiplication, etc.) and with more sophisticated operations (trigonometric functions, exponential and logarithmic functions, etc.). Pandas inherits much of this functionality from NumPy, and the *universal functions* (ufuncs for short) which we introduced in section X.X are key to this.

Pandas includes a couple useful twists, however: for unary operations like negation and trigonometric functions, these ufuncs will *preserve index and column labels* in the output, and for binary operations such as addition and multiplication, Pandas will automatically *align indices* when passing the objects to the ufunc. This means that keeping the context of data, and combining data from different sources – both potentially error-prone tasks with raw NumPy arrays – become essentially foolproof with Pandas. We will additionally see that there are well-defined operations between one-dimensional Series structures and two-dimensional DataFrame structures.

Ufuncs: Index Preservation

Because Pandas is designed to work with NumPy, any NumPy ufunc will work on pandas Series and DataFrame objects. Lets start by defining a simple Series and DataFrame on which to demonstrate this:

```
import pandas as pd
import numpy as np

rng = np.random.RandomState(42)
ser = pd.Series(rng.randint(0, 10, 4))
ser
0    6
1    3
2    7
3    4
dtype: int64

df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                  columns=['A', 'B', 'C', 'D'])
df
   A  B  C  D
```

```

0  6  9  2  6
1  7  4  3  7
2  7  2  5  4

```

If we apply a NumPy ufunc on either of these objects, the result will be another Pandas object *with the indices preserved*:

```

np.exp(ser)
0      403.428793
1       20.085537
2    1096.633158
3       54.598150
dtype: float64

```

Or, for a slightly more complex calculation:

```

np.sin(df * np.pi / 4)

```

	A	B	C	D
0	-1.000000	7.071068e-01	1.000000	-1.000000e+00
1	-0.707107	1.224647e-16	0.707107	-7.071068e-01
2	-0.707107	1.000000e+00	-0.707107	1.224647e-16

Any of the ufuncs discussed in Section X.X can be used in a similar manner.

UFuncs: Index Alignment

For binary operations on two Series or DataFrame objects, Pandas will align indices in the process of performing the operation. This is very convenient when working with incomplete data, as we'll see in some of the examples below.

Index Alignment in Series

As an example, suppose we are combining two different data sources, and find only the top three US states by *area* and the top three US states by *population*:

```

area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                  'California': 423967}, name='area')
population = pd.Series({'California': 38332521, 'Texas': 26448193,
                       'New York': 19651127}, name='population')

```

Let's see what happens when we divide these to compute the population density:

```

population / area
Alaska      NaN
California   90.413926
New York     NaN
Texas        38.018740
dtype: float64

```

The resulting array contains the *union* of indices of the two input arrays, which could be determined using standard Python set arithmetic on these indices:

```
area.index | population.index
Index(['Alaska', 'California', 'New York', 'Texas'], dtype='object')
```

Any item for which one or the other does not have an entry is marked by NaN, or “Not a Number”, which is how Pandas marks missing data (see further discussion of missing data in Section X.X). This index matching is implemented this way for any of Python’s built-in arithmetic expressions; any missing values are filled-in with NaN by default:

```
A = pd.Series([2, 4, 6], index=[0, 1, 2])
B = pd.Series([1, 3, 5], index=[1, 2, 3])
A + B
0    NaN
1     5
2     9
3    NaN
dtype: float64
```

If filling-in NaN values is not the desired behavior, the fill value can be modified using appropriate object methods in place of the operators. For example, calling `A.add(B)` is equivalent to calling `A + B`, but allows optional explicit specification of the fill value:

```
A.add(B, fill_value=0)
0     2
1     5
2     9
3     5
dtype: float64
```

Index Alignment in DataFrame

A similar type of alignment takes place for *both* columns and indices when performing operations on dataframes:

```
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
                  columns=list('AB'))
A
   A  B
0  1  11
1  5   1

B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
                  columns=list('BAC'))
B
   B  A  C
0  4  0  9
1  5  8  0
2  9  2  6

A + B
   A  B  C
0  1  15 NaN
```

```

1  13    6 NaN
2 NaN NaN NaN

```

Notice that indices are aligned correctly irrespective of their order in the two objects, and indices in the result are sorted. Similarly to the case of the Series, we can use the associated object's arithmetic method and pass any desired `fill_value` to be used in place of missing entries:

```

A.add(B, fill_value=np.mean(A.values))

```

	A	B	C
0	1.0	15.0	13.5
1	13.0	6.0	4.5
2	6.5	13.5	10.5

A table of Python operators and their equivalent Pandas object methods follows:

Operator	Pandas Method(s)
+	<code>add()</code>
-	<code>sub()</code> , <code>subtract()</code>
*	<code>mul()</code> , <code>multiply()</code>
/	<code>truediv()</code> , <code>div()</code> , <code>divide()</code>
//	<code>floordiv()</code>
%	<code>mod()</code>
**	<code>pow()</code>

Ufuncs: Operations between DataFrame and Series

When performing operations between a `DataFrame` and a `Series`, the index and column alignment is similarly maintained. Operations between a `DataFrame` and a `Series` are similar to operations between a 2D and 1D NumPy array. Consider one common operation, where we find the difference of a 2D array and one of its rows:

```

A = rng.randint(10, size=(3, 4))
A
array([[3, 8, 2, 4],
       [2, 6, 4, 8],
       [6, 1, 3, 8]])

A - A[0]
array([[ 0,  0,  0,  0],
       [-1, -2,  2,  4],
       [ 3, -7,  1,  4]])

```

According to NumPy's broadcasting rules (see Section X.X), subtraction between a two-dimensional array and one of its rows is applied row-wise.

In Pandas, the convention similarly operates row-wise by default:

```
df = pd.DataFrame(A, columns=list('QRST'))
df - df.iloc[0]
   Q  R  S  T
0  0  0  0  0
1 -1 -2  2  4
2  3 -7  1  4
```

If you would instead like to operate column-wise, you can use the object methods mentioned above, while specifying the axis keyword:

```
df.subtract(df['R'], axis=0)
   Q  R  S  T
0 -5  0 -6 -4
1 -4  0 -2  2
2  5  0  2  7
```

Note that these DataFrame/Series operations, like the operations discussed above, will automatically align indices between the two elements:

```
halfrow = df.iloc[0, ::2]
halfrow
Q    3
S    2
Name: 0, dtype: int64

df - halfrow
   Q  R  S  T
0  0 NaN  0 NaN
1 -1 NaN  2 NaN
2  3 NaN  1 NaN
```

This preservation and alignment of indices and columns means that operations on data in Pandas will always maintain the data context, which prevents the types of silly errors that might come up when working with heterogeneous data in raw NumPy arrays.

Summary

We've shown that standard NumPy ufuncs will operate element-by-element on Pandas objects, with some additional useful functionality: they preserve index and column names, and automatically align different sets of indices and columns. Like the basic indexing and selection operations we saw in the previous section, these types of element-wise operations on Series and DataFrames form the building blocks of many more sophisticated data processing examples to come. The index alignment operations, in particular, sometimes lead to a state where values are missing from the resulting arrays. In the next section we will discuss in detail how Pandas chooses to handle such missing values.

Handling Missing Data

The difference between data found in many tutorials and data in the real world is that real-world data is rarely clean and homogeneous. In particular, many interesting datasets will have some amount of data missing. To make matters even more complicated, different data sources may indicate missing data in different ways.

In this section, we will discuss some general considerations for missing data, discuss how Pandas chooses to represent it, and demonstrate some built-in Pandas tools for handling missing data in Python. Here and throughout the book, we'll refer to missing data in general as “null”, “NaN”, or “NA” values.

Tradeoffs in Missing Data Conventions

There are a number of schemes that have been developed to indicate the presence of missing data in an array of data. Generally, they revolve around one of two strategies: using a *mask* which globally indicates missing values, or choosing a *sentinel value* which indicates a missing entry.

In the masking approach, the mask might be an entirely separate boolean array, or it may involve appropriation of one bit in the data representation to locally indicate the null status of a value.

In the sentinel approach, the sentinel value could be some data-specific convention, such as indicating a missing integer value with -9999 or some rare bit pattern, or it could be a more global convention, such as indicating a missing floating point value with NaN (Not a Number), a special value which is part of the IEEE floating point specification.

None of these approaches is without tradeoffs: use of a separate mask array requires allocation of an additional boolean array which adds overhead in both storage and computation. A sentinel value reduces the range of valid values which can be represented, and may require extra (often non-optimized) logic in CPU & GPU arithmetic. Common special values like NaN are not available for all data types.

As in most cases where no universally optimal choice exists, different languages and systems use different conventions. For example, the R language uses reserved bit patterns within each data type as sentinel values indicating missing data, while the SciDB system uses an extra byte attached to every cell which indicates a NA state.

Missing Data in Pandas

Pandas' choice for how to handle missing values is constrained by its reliance on the NumPy package, which does not have a built-in notion of NA values for non-floating-point datatypes.

Pandas could have followed R's lead in specifying bit patterns for each individual data type to indicate nullness, but this approach turns out to be rather unwieldy in Pandas' case. While R contains four basic data types, NumPy supports *far* more than this: for example, while R has a single integer type, NumPy supports *fourteen* basic integer types once you account for available precisions, signedness, and endianness of the encoding. Reserving a specific bit pattern in all available NumPy types would lead to an unwieldy amount of overhead in special-casing various operations for various types, and the implementation would probably require a new fork of the NumPy package.

NumPy does have support for masked arrays – i.e. arrays which have a separate boolean mask array attached which marks data as “good” or “bad”. Pandas could have derived from this, but the overhead in both storage, computation, and code maintenance makes that an unattractive choice.

With these constraints in mind, Pandas chose to use sentinels for missing data, and further chose to use two already-existing Python null values: the special floating-point NaN value, and the Python None object. This choice has some side-effects, as we will see, but in practice ends up being a good compromise in most cases of interest.

None: Pythonic Missing Data

The first sentinel value used by Pandas is None. None is a Python singleton object which is often used for missing data in Python code. Because it is a Python object, it cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type 'object' (i.e. arrays of Python objects):

```
import numpy as np
import pandas as pd

vals1 = np.array([1, None, 3, 4])
vals1
array([1, None, 3, 4], dtype=object)
```

This dtype=object means that the best common type representation NumPy could infer for the contents of the array is that they are Python objects. While this kind of object array is useful for some purposes, any operations on the data will be done at the Python level, with much more overhead than the typically fast operations seen for arrays with native types.

```
for dtype in ['object', 'int']:
    print("dtype =", dtype)
    %timeit np.arange(1E6, dtype=dtype).sum()
    print()
dtype = object
10 loops, best of 3: 73.3 ms per loop
```

```
dtype = int
100 loops, best of 3: 3.08 ms per loop
```

The use of Python objects in an array also means that if you perform aggregations like `sum()` or `min()` across an array with a `None` value, you will generally get an error.

```
vals1.sum()
```

This is because addition in Python between an integer and `None` is undefined.

NaN: Missing Numerical Data

The other missing data representation, `NaN` (acronym for *Not a Number*) is different: it is a special floating-point value that is recognized by all systems which use the standard IEEE floating-point representation.

```
vals2 = np.array([1, np.nan, 3, 4])
vals2.dtype
dtype('float64')
```

Notice that NumPy chose a native floating-point type for this array: this means that unlike the object array above, this array supports fast operations pushed into compiled code. You should be aware that `NaN` is a bit like a data virus which infects any other object it touches. Regardless of the operation, the result of arithmetic with `NaN` will be another `NaN`:

```
1 + np.nan
nan

0 * np.nan
nan
```

Note that this means that the sum or maximum of the values is well-defined (it doesn't result in an error), but not very useful:

```
vals2.sum(), vals2.min(), vals2.max()
(nan, nan, nan)
```

Keep in mind that `NaN` is specifically a floating-point value; there is no equivalent `NaN` value for integers, strings, or other types.

Examples

Each of the above sentinel representations has its place, and Pandas is built to handle the two of them nearly interchangeably, and will convert between the two sentinel values where appropriate:

```
data = pd.Series([1, np.nan, 2, None])
data
0      1
1    NaN
2      2
```



```
3    NaN
dtype: float64
```

Keep in mind, though, that because `None` is a Python object type and `NaN` is a floating-point type, there is *no in-type NA representation in Pandas for string, boolean, or integer values*. Pandas gets around this by type-casting in cases where NA values are present. For example, if we set a value in an integer array to `np.nan`, it will automatically be up-cast to a floating point type to accommodate the NA:

```
x = pd.Series(range(2), dtype=int)
x[0] = None
x
0    NaN
1     1
dtype: float64
```

Notice that in addition to casting the integer array to floating point, Pandas automatically converts the `None` to a `NaN` value. Though this type of magic may feel a bit hackish compared to the more unified approach to NA values in domain-specific languages like R, the Pandas sentinel/casting approach works well in practice and in my experience only rarely causes issues.

Here is a short table of the upcasting conventions in Pandas when NA values are introduced:

Typeclass	Promotion when storing NAs	NA sentinel value
floating	no change	<code>np.nan</code>
object	no change	<code>None</code> or <code>np.nan</code>
integer	cast to float64	<code>np.nan</code>
boolean	cast to object	<code>None</code> or <code>np.nan</code>

Keep in mind that in Pandas, string data is always stored with an object dtype.

Operating on Null Values

As we have seen, Pandas treats `None` and `NaN` as essentially interchangeable for indicating missing or null values. To facilitate this convention, there are several useful methods for detecting, removing, and replacing null values in Pandas data structures. They are:

- `isnull()`: generate a boolean mask indicating missing values
- `notnull()`: opposite of `isnull()`
- `dropna()`: return a filtered version of the data
- `fillna()`: return a copy of the data with missing values filled or imputed

We will finish this section with a brief discussion and demonstration of these routines:

Detecting Null Values

Pandas data structures have two useful methods for detecting null data: `isnull()` and `notnull()`. Either one will return a boolean mask over the data, for example:

```
data = pd.Series([1, np.nan, 'hello', None])

data.isnull()
0    False
1     True
2    False
3     True
dtype: bool
```

As mentioned in section X.X, boolean masks can be used directly as a Series or DataFrame index:

```
data[data.notnull()]
0    1
2  hello
dtype: object
```

The `isnull()` and `notnull()` methods produce similar boolean results for DataFrames.

Dropping Null Values

In addition to the masking used above, there are the convenience methods, `dropna()` and `fillna()`, which respectively remove NA values and fill-in NA values. For a Series, the result is straightforward:

```
data.dropna()
0    1
2  hello
dtype: object
```

For a dataframe, there are more options. Consider the following dataframe:

```
df = pd.DataFrame([[1,      np.nan, 2],
                   [2,      3,      5],
                   [np.nan, 4,      6]])

df
   0  1  2
0  1 NaN 2
1  2  3  5
2 NaN  4  6
```

We cannot drop single values from a DataFrame; we can only drop full rows or full columns. Depending on the application, you might want one or the other, so `dropna()` gives a number of options for a DataFrame.

By default, `dropna()` will drop all rows in which *any* null value is present:

```
df.dropna()
   0  1  2
1  2  3  5
```

Alternatively, you can drop NA values along a different axis: `axis=1` drops all columns containing a null value:

```
df.dropna(axis=1)
   2
0  2
1  5
2  6
```

But this drops some good data as well; you might rather be interested in dropping rows or columns with *all* NA values, or a majority of NA values. This can be specified through the `how` or `thresh` parameters, which allow fine control of the number of nulls to allow through.

The default is `how='any'`, such that any row or column (depending on the `axis` keyword) containing a null value will be dropped. You can also specify `how='all'`, which will only drop rows/columns which are *all* null values:

```
df[3] = np.nan
df
   0  1  2  3
0  1 NaN  2 NaN
1  2  3  5 NaN
2 NaN  4  6 NaN

df.dropna(axis=1, how='all')
   0  1  2
0  1 NaN  2
1  2  3  5
2 NaN  4  6
```

Keep in mind that to be a bit more clear, you can use `axis='rows'` rather than `axis=0` and `axis='columns'` rather than `axis=1`.

For finer-grained control, the `thresh` parameter lets you specify a minimum number of non-null values for the row/column to be kept:

```
df.dropna(thresh=3)
   0  1  2  3
1  2  3  5 NaN
```

Here the first and last row have been dropped, because they contain only two non-null values.

Filling Null Values

Sometimes rather than dropping NA values, you'd rather replace them with a valid value. This value might be a single number like zero, or it might be some sort of imputation or interpolation from the good values. You could do this in-place using the `isnull()` method as a mask, but because it is such a common operation Pandas provides the `fillna()` method, which returns a copy of the array with the null values replaced.

Consider the following Series:

```
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
data
a      1
b    NaN
c      2
d    NaN
e      3
dtype: float64
```

We can fill NA entries with a single value, such as zero:

```
data.fillna(0)
a      1
b      0
c      2
d      0
e      3
dtype: float64
```

We can specify a forward-fill to propagate the previous value forward:

```
# forward-fill
data.fillna(method='ffill')
a      1
b      1
c      2
d      2
e      3
dtype: float64
```

Or we can specify a back-fill to propagate the next values backward:

```
# back-fill
data.fillna(method='bfill')
a      1
b      2
c      2
d      3
```

```
e    3
dtype: float64
```

For DataFrames, the options are similar, but we can also specify an axis along-which the fills take place:

```
df
   0  1  2  3
0  1 NaN 2 NaN
1  2  3 5 NaN
2 NaN  4 6 NaN

df.fillna(method='ffill', axis=1)
   0  1  2  3
0  1  1  2  2
1  2  3  5  5
2 NaN  4  6  6
```

Notice that if a previous value is not available during a forward fill, the NA value remains.

Summary

Here we have seen how Pandas handles null/NA values, and seen a few DataFrame and Series methods specifically designed to handle these missing values in a uniform way. Missing data is a fact of life in real-world datasets, and we'll see these tools often in the following chapters.

Hierarchical Indexing

Up to this point we've been focused primarily on one-dimensional and two-dimensional data, stored in Pandas Series and DataFrame objects respectively. But sometimes you'd like to go beyond this and store higher-dimensional data, that is, data indexed by more than one or two keys. While Pandas does provide Panel and Panel4D objects which natively handle 3D and 4D data (see below), a far more common pattern in practice is to make use of *hierarchical indexing* (also known as *multi-indexing*) to incorporate multiple index *levels* within a single index. In this way, higher-dimensional data can be compactly represented within the familiar one-dimensional Series and two-dimensional DataFrame objects.

In this section we'll explore the direct creation of MultiIndex objects, considerations when indexing, slicing, and computing statistics across multiply-indexed data, and useful routines for converting between simple and hierarchically-indexed representations of your data.

```
import pandas as pd
import numpy as np
```

A Multiply-Indexed Series

Let's start by considering how we might represent two-dimensional data within a one-dimensional Series. For concreteness, consider a series of data where each point has a character and numerical key.

The Bad Way...

Consider a case in which you want to track data about states in two different years. Using the Pandas tools we've already covered, you might be tempted to simply use Python tuples as keys:

```
index = [('California', 2000), ('California', 2010),
         ('New York', 2000), ('New York', 2010),
         ('Texas', 2000), ('Texas', 2010)]
populations = [33871648, 37253956,
               18976457, 19378102,
               20851820, 25145561]
pop = pd.Series(populations, index=index)
pop
(California, 2000)    33871648
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
(Texas, 2010)         25145561
dtype: int64
```

With this indexing scheme, you can straightforwardly index or slice the series based on this multiple index:

```
pop[('California', 2010):('Texas', 2000)]
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
dtype: int64
```

but the convenience ends there. If you wish, for example, to select all 2010 values, you'll need to do some messy and slow munging to make it happen:

```
pop[[i for i in pop.index if i[1] == 2010]]
(California, 2010)    37253956
(New York, 2010)      19378102
(Texas, 2010)         25145561
dtype: int64
```

This produces the desired result, but is not as clean (or as efficient for large datasets) as the slicing syntax we've grown to love in Pandas.

The Better Way... Pandas MultiIndex

Fortunately, Pandas provides a better way. Our tuple-based indexing is essentially a poor-man's multi-index, and the Pandas `MultiIndex` type gives us the type of operations we wish to have. We can create a multi-index from the tuples as follows:

```
mindex = pd.MultiIndex.from_tuples(index)
mindex
MultiIndex(levels=[['California', 'New York', 'Texas'], [2000, 2010]],
            labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

Notice that the `MultiIndex` contains multiple *levels* of indexing, in this case the state names and the years, as well as multiple *labels* for each data point which encode these levels.

If we re-index our series with this `MultiIndex`, we see the Hierarchical representation of the data:

```
pop = pop.reindex(mindex)
pop
California 2000    33871648
           2010    37253956
New York   2000    18976457
           2010    19378102
Texas      2000    20851820
           2010    25145561
dtype: int64
```

Here the first two columns of the Series representation show the multiple index values, while the third column shows our data. Notice that some entries are missing in the first column: in this multi-index representation, any blank entry indicates the same value as the line above it.

Now to access all data for which the second index is 2010, we can simply use Pandas' slicing notation:

```
pop[:, 2010]
California    37253956
New York      19378102
Texas         25145561
dtype: int64
```

The result is a singly-indexed array with just the keys we're interested in. This syntax is much more convenient (and the operation is much more efficient!) than the home-spun tuple-based multi-indexing solution that we started with. Below we'll further discuss this sort of indexing operation on hierarchically indexed data.

MultiIndex as Extra Dimension

The astute reader might notice something else here: we could easily have stored the same data using a simple dataframe with index and column labels. In fact, Pandas is

built with this equivalence in mind. The `unstack()` method will quickly convert a multiply-indexed Series into a conventionally-indexed DataFrame:

```
pop_df = pop.unstack()
pop_df
```

	2000	2010
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

Naturally, the `stack()` method provides the opposite operation:

```
pop_df.stack()
California 2000    33871648
           2010    37253956
New York   2000    18976457
           2010    19378102
Texas      2000    20851820
           2010    25145561
dtype: int64
```

Seeing this, you might wonder why would we would bother with hierarchical indexing at all. The reason is simple: just as we were able to use multi-indexing to represent two-dimensional data within a one-dimensional Series, we can also use it to represent three or higher-dimensional data in a Series or DataFrame. Each extra level in a multi-index represents an extra dimension of data; taking advantage of this property gives us much more flexibility in the types of data we can represent. Concretely, we might want to add another column of demographic (say, population under 18) data for each state at each year; with a multiindex this is as easy as adding another column to the dataframe.

```
pop_df = pd.DataFrame({'total': pop,
                       'under18': [9267089, 9284094,
                                   4687374, 4318033,
                                   5906301, 6879014]})
pop_df
```

		total	under18
California	2000	33871648	9267089
	2010	37253956	9284094
New York	2000	18976457	4687374
	2010	19378102	4318033
Texas	2000	20851820	5906301
	2010	25145561	6879014

In addition, all the ufuncs and other functionality discussed in Section X.X work with hierarchical indices as well:

```
f_u18 = pop_df['under18'] / pop_df['total']
print("fraction of population under 18:")
f_u18.unstack()
fraction of population under 18:
2000    2010
```


California	0.273594	0.249211
New York	0.247010	0.222831
Texas	0.283251	0.273568

This allows us to easily and quickly manipulate and explore even high-dimensional data.

Aside: Panel Data

Pandas has a few other fundamental data structures that we have not yet discussed, namely the `pd.Panel` and `pd.Panel4D` objects. These can be thought of, respectively, as three-dimensional and four-dimensional generalizations of the (one-dimensional) `Series` and (two-dimensional) `DataFrame` structures. Once you are familiar with indexing and manipulation of data in a `Series` and `DataFrame`, the `Panel` and `Panel4D` are relatively straightforward to use. In particular, the `ix`, `loc`, and `iloc` indexers discussed in section X.X extend readily to these higher-dimensional structures.

We won't cover these panel structures further in this text, as I've found in the majority of cases that multi-indexing is a more useful and conceptually simple representation for higher-dimensional data. Additionally, panel data is fundamentally a dense data representation, while multiindexing is fundamentally a sparse data representation. As the number of dimensions grows, the dense representation becomes very inefficient for the majority of real-world datasets. For the occasional specialized application, however, these structures can be useful. If you'd like to read more about the `Panel` and `Panel4D` structures, see the references listed in section X.X.

Methods of MultiIndex Creation

The most straightforward way to construct a multiply-indexed `Series` or `DataFrame` is to simply pass a list of two index arrays to the constructor. For example:

```
df = pd.DataFrame(np.random.rand(4, 2),
                  index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                  columns=['data1', 'data2'])

df
```

		data1	data2
a	1	0.111677	0.887069
	2	0.195157	0.782505
b	1	0.921233	0.912892
	2	0.493172	0.736983

The work of creating the `MultiIndex` is done in the background.

Similarly, if you pass a dictionary with appropriate tuples as keys, Pandas will automatically recognize this and use a `MultiIndex` by default:

```
data = {('California', 2000): 33871648,
        ('California', 2010): 37253956,
        ('Texas', 2000): 20851820,
```

```

        ('Texas', 2010): 25145561,
        ('New York', 2000): 18976457,
        ('New York', 2010): 19378102}
pd.Series(data)
California 2000    33871648
           2010    37253956
New York   2000    18976457
           2010    19378102
Texas      2000    20851820
           2010    25145561
dtype: int64

```

Nevertheless, it is sometimes useful to explicitly create a MultiIndex; we'll see a couple of these methods below.

Explicit MultiIndex Constructors

For more flexibility in how the index is constructed, you can instead use the class method constructors available in the `pd.MultiIndex`. For example, in analogy to above, you can construct the MultiIndex from a simple list of arrays giving the index values within each level:

```

pd.MultiIndex.from_arrays([[ 'a', 'a', 'b', 'b'], [1, 2, 1, 2]])
MultiIndex(levels=[[ 'a', 'b'], [1, 2]],
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

```

You can construct it from a list of tuples giving the multiple index values of each point:

```

pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])
MultiIndex(levels=[[ 'a', 'b'], [1, 2]],
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

```

You can even construct it from a Cartesian product of unique index values:

```

pd.MultiIndex.from_product([[ 'a', 'b'], [1, 2]])
MultiIndex(levels=[[ 'a', 'b'], [1, 2]],
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

```

Similarly, you can construct the MultiIndex directly using its internal encoding by passing `levels`, a list of lists containing available index values for each level, and `labels`, a list of lists which reference these labels:

```

pd.MultiIndex(levels=[[ 'a', 'b'], [1, 2]],
              labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
MultiIndex(levels=[[ 'a', 'b'], [1, 2]],
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

```

Any of these objects can be passed as the `index` argument when creating a Series or DataFrame, or be passed to the `reindex` method of an existing Series or DataFrame.

MultiIndex Level Names

Sometimes it is convenient to name the levels of the MultiIndex. This can be accomplished by passing the `names` argument to any of the above MultiIndex constructors, or by setting the `names` attribute of the index after the fact:

```
pop.index.names = ['state', 'year']
pop
state      year
California 2000    33871648
           2010    37253956
New York   2000    18976457
           2010    19378102
Texas      2000    20851820
           2010    25145561
dtype: int64
```

With more involved data sets, this can be a useful way to keep track of the meaning of various index values.

MultiIndex for Columns

In a DataFrame, the rows and columns are completely symmetric, and just as the rows can have multiple levels of indices, the columns can have multiple levels as well. Consider the following, which is a mock-up of some (somewhat realistic) medical data:

```
# hierarchical indices and columns
index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]],
                                   names=['year', 'visit'])
columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'], ['HR', 'Temp']],
                                    names=['subject', 'type'])

# mock some data
data = np.round(np.random.randn(4, 6), 1)
data[:, :2] *= 10
data += 37

# create the dataframe
health_data = pd.DataFrame(data, index=index, columns=columns)
health_data
```

subject		Bob		Guido		Sue	
type		HR	Temp	HR	Temp	HR	Temp
year visit							
2013	1	42	35.3	35	36.1	43	36.1
	2	12	37.7	38	37.0	33	35.8
2014	1	44	37.4	53	37.0	37	36.7
	2	46	37.9	26	36.0	27	35.3

Here we see where the multi-indexing for both rows and columns can come in *very* handy. This is fundamentally four-dimensional data, where the dimensions are the

subject, the measurement type, the year, and the visit number. With this in place we can, for example, index the top level column by the person's name and get a full DataFrame containing just that person's information:

```
health_data['Guido']
type      HR  Temp
year visit
2013  1    35  36.1
      2    38  37.0
2014  1    53  37.0
      2    26  36.0
```

For complicated records containing multiple labeled measurements across multiple times for many subjects (be they people, countries, cities, etc.) use of Hierarchical rows and columns can be extremely convenient!

Indexing and Slicing a MultiIndex

Indexing and slicing on a MultiIndex is designed to be intuitive, and it helps if you think about the indices as added dimensions. We'll first look at indexing multiply-indexed Series, and then multiply-indexed DataFrames.

Multiply-indexed Series

Consider the multiply-indexed series of state populations we saw above:

```
pop
state  year
California  2000    33871648
           2010    37253956
New York   2000    18976457
           2010    19378102
Texas      2000    20851820
           2010    25145561
dtype: int64
```

We can access single elements by indexing with multiple terms:

```
pop['California', 2000]
33871648
```

The MultiIndex also supports *partial indexing*, or indexing just one of the levels in the index. The result is another Series, with the lower-level indices maintained:

```
pop['California']
year
2000    33871648
2010    37253956
dtype: int64
```

Partial slicing is available as well, as long as the MultiIndex is sorted (see discussion below):

```
pop.loc['California':'New York']
state      year
California 2000    33871648
           2010    37253956
New York   2000    18976457
           2010    19378102
dtype: int64
```

With sorted indices, partial indexing can be performed on lower levels by passing an empty slice in the first index:

```
pop[:, 2000]
state
California    33871648
New York      18976457
Texas         20851820
dtype: int64
```

Other types of indexing and selection discussed in section X.X work as well; for example selection based on boolean masks:

```
pop[pop > 22000000]
state      year
California 2000    33871648
           2010    37253956
Texas      2010    25145561
dtype: int64
```

Selection based on fancy indexing works as well:

```
pop[['California', 'Texas']]
state      year
California 2000    33871648
           2010    37253956
Texas      2000    20851820
           2010    25145561
dtype: int64
```

Multiply-indexed DataFrames

A multiply-indexed DataFrame behaves in a similar manner. Consider our toy medial dataframe from above:

```
health_data
subject  Bob      Guido      Sue
type     HR  Temp  HR  Temp  HR  Temp
year visit
2013  1    42  35.3   35  36.1  43  36.1
      2    12  37.7   38  37.0  33  35.8
2014  1    44  37.4   53  37.0  37  36.7
      2    46  37.9   26  36.0  27  35.3
```

Remember that columns are primary in a dataframe, and the syntax used for multiply indexed Series above applies to the columns. For example, we can recover Guido's heartrate data with a simple operation:

```
health_data['Guido', 'HR']
year visit
2013 1      35
      2      38
2014 1      53
      2      26
Name: (Guido, HR), dtype: float64
```

Also similarly to the single-index case, we can use the `loc`, `iloc`, and `ix` indexers introduced in section X.X. For example:

```
health_data.iloc[:2, :2]
subject      Bob
type         HR  Temp
year visit
2013 1      42  35.3
      2      12  37.7
```

these indexers provide an array-like view of the underlying two-dimensional data, but each individual index in `loc` or `iloc` can be passed a tuple of multiple indices. For example:

```
health_data.loc[:, ('Bob', 'HR')]
year visit
2013 1      42
      2      12
2014 1      44
      2      46
Name: (Bob, HR), dtype: float64
```

Working with slices within these index tuples is not especially convenient; trying to create a slice within a tuple will lead to a syntax error:

```
health_data.loc[:, 1), (:, 'HR')]
```

You could get around this by building the desired slice explicitly using Python's built-in `slice()` function, but a better way in this context is to use Pandas `IndexSlice` object, which is provided for precisely this situation. For example:

```
idx = pd.IndexSlice
health_data.loc[idx[:, 1], idx[:, 'HR']]
subject      Bob Guido Sue
type         HR   HR   HR
year visit
2013 1      42    35   43
2014 1      44    53   37
```

There are so many ways to interact with data in multiply-indexed Series and DataFrames, and as with many tools in this book the best way to become familiar with them is to try them out!

Rearranging Multi-Indices

One of the keys to effectively working with multiply-indexed data is knowing how to effectively transform the data. There are a number of operations which will preserve all the information in the dataset, but rearrange it for the purposes of various computations. We saw a brief example of this in the `stack()` and `unstack()` methods above, but there are many more ways to finely control the rearrangement of data between hierarchical indices and columns, and we'll explore them here.

Sorted and Unsorted Indices

We briefly mentioned a caveat above, but we should emphasize it more here. **Many of the MultiIndex slicing operations will fail if the index is not sorted.** Let's take a look at this here.

We'll start by creating some simple multiply-indexed data where the indices are *not lexographically sorted*:

```
index = pd.MultiIndex.from_product(['a', 'c', 'b'], [1, 2])
data = pd.Series(np.random.rand(6), index=index)
data.index.names = ['char', 'int']
data
char  int
a      1    0.921624
      2    0.280299
c      1    0.459172
      2    0.586443
b      1    0.252360
      2    0.227359
dtype: float64
```

If we try to take a partial slice of this index, it will result in an error. For clarity here, we'll catch the error and print the message:

```
try:
    data['a':'b']
except KeyError as e:
    print(type(e))
    print(e)
<class 'KeyError'>
'Key length (1) was greater than MultiIndex lexsort depth (0)'
```

Though it is not entirely clear from the error message, this is the result of the fact that the MultiIndex is not sorted. For various reasons, partial slices and other similar operations require the levels in the MultiIndex to be in sorted (i.e. lexographical)

order. Pandas provides a number of convenience routines to perform this type of sorting; examples are the `sort_index()` and `sortlevel()` methods of the dataframe. We'll use the simplest, `sort_index()`, here:

```
data = data.sort_index()
data
char  int
a     1    0.921624
      2    0.280299
b     1    0.252360
      2    0.227359
c     1    0.459172
      2    0.586443
dtype: float64
```

With the index sorted in this way, partial slicing will work as expected:

```
data['a':'b']
char  int
a     1    0.921624
      2    0.280299
b     1    0.252360
      2    0.227359
dtype: float64
```

Stacking and Unstacking Indices

As we saw briefly above, it is possible to convert a dataset from a stacked multi-index to a simple two-dimensional representation:

```
pop_df = pop.unstack()
pop_df
year          2000          2010
state
California  33871648  37253956
New York    18976457  19378102
Texas       20851820  25145561
```

The opposite of unstacking is stacking:

```
pop_df.stack()
state  year
California  2000    33871648
          2010    37253956
New York   2000    18976457
          2010    19378102
Texas      2000    20851820
          2010    25145561
dtype: int64
```

Each of these methods has keywords which can control the ordering of levels and dimensions in the output; see the method documentation for details.

Index Setting and Resetting

Another way to rearrange hierarchical data is to turn the index labels into columns; this can be accomplished with the `reset_index` method. Calling this on the population dictionary will result in a series with a *state* and *year* column holding the information that was formerly in the index. For clarity, we can optionally specify the name of the data for the column representation:

```
pop_flat = pop.reset_index(name='population')
pop_flat
```

	state	year	population
0	California	2000	33871648
1	California	2010	37253956
2	New York	2000	18976457
3	New York	2010	19378102
4	Texas	2000	20851820
5	Texas	2010	25145561

Often when working with data in the real world, the raw input data looks like this and it's useful to build a MultiIndex from the column values. This can be done with the `set_index` method of the DataFrame, which returns a Multiply-indexed DataFrame:

```
pop_flat.set_index(['state', 'year'])
```

		population
state	year	
California	2000	33871648
	2010	37253956
New York	2000	18976457
	2010	19378102
Texas	2000	20851820
	2010	25145561

In practice, I find this to be one of the most useful patterns with real-world datasets.

Data Aggregations on Multi-Indices

We've previously seen that Pandas has built-in data aggregation methods, such as `mean()`, `sum()`, `max()`, etc. For hierarchically indexed data, these can be passed a `level` parameter which controls which subset of the data the aggregate is computed on.

For example, let's return to our health data:

```
health_data
```

subject		Bob		Guido		Sue	
type		HR	Temp	HR	Temp	HR	Temp
year visit							
2013	1	42	35.3	35	36.1	43	36.1
	2	12	37.7	38	37.0	33	35.8
2014	1	44	37.4	53	37.0	37	36.7
	2	46	37.9	26	36.0	27	35.3

Perhaps we'd like to average-out the measurements in the two visits each year. We can do this by naming the index level we'd like to explore, in this case the year:

```
data_mean = health_data.mean(level='year')
data_mean
subject Bob      Guido      Sue
type    HR    Temp    HR    Temp    HR    Temp
year
2013     27  36.50  36.5  36.55  38   35.95
2014     45  37.65  39.5  36.50  32   36.00
```

by further making use of the `axis` keyword, we can take the mean among levels on the columns as well:

```
data_mean.mean(axis=1, level='type')
type      HR      Temp
year
2013  33.833333  36.333333
2014  38.833333  36.716667
```

Thus in two lines, we've been able to find the average heart rate and temperature measured among all subjects in all visits each year. While this is a toy example, many real-world datasets have similar hierarchical structure.

Summary

In this section we've covered many aspects of Hierarchical Indexing or Multi-Indexing, which is a convenient way to store, manipulate, and process multi-dimensional data. Probably the most important skill involved in this is the ability to transform data from hierarchical representations to flat representations to multi-dimensional representations. You'll find in working with real datasets that these transformation operations are often the first step in preparing data for analysis. In a later section we'll see some examples of these tools in action.

Combining Datasets: Concat & Append

Some of the most interesting studies of data come from combining different data sources. These operations can involve anything from very straightforward concatenation of two different datasets, to more complicated database-style joins and merges which correctly handle indices which might be shared between the datasets. Pandas series and dataframes are built with this type of operation in mind, and Pandas includes functions and methods which make this sort of data wrangling fast and straightforward.

In this section we'll take a look at simple concatenation of Series and DataFrames with the `pd.concat` function; in the following section we'll take a look at the more sophisticated in-memory merges and joins implemented in Pandas.

```
import pandas as pd
import numpy as np
```

For convenience, we'll define this function which creates a DataFrame of a particular form that will be useful below:

```
def make_df(cols, ind):
    """Quickly make a dataframe"""
    data = {c: [str(c) + str(i) for i in ind]
             for c in cols}
    return pd.DataFrame(data, ind)
```

```
# example DataFrame
make_df('ABC', range(3))
   A  B  C
0 A0 B0 C0
1 A1 B1 C1
2 A2 B2 C2
```

In addition, we'll create a quick class which allows us to display multiple dataframes side-by-side. The code makes use of the special `_repr_html_` method, which IPython uses to implement its rich object display:

```
class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style='float: left; padding: 10px;'>
    <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                          for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                            for a in self.args)
```

The use of this will become more clear below!

Recall: Concatenation of NumPy Arrays

Concatenation of Series and DataFrames is very similar to concatenation of Numpy arrays, which can be done via the `np.concatenate` function as discussed in Section X.X. Recall that with it, you can combine the contents of two arrays into a single array:

```
x = [1, 2, 3]
y = [4, 5, 6]
z = [7, 8, 9]
```

```
np.concatenate([x, y, z])
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

The first argument is a list or tuple of arrays to concatenate. Additionally, it takes an `axis` keyword that allows you to specify the axis along which the result will be concatenated:

```
x = [[1, 2],
      [3, 4]]
np.concatenate([x, x], axis=1)
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
```

Simple Concatenation with `pd.concat`

Pandas has a similar function, `pd.concat()`, which has a similar syntax, but which contains a number of options that we'll discuss below:

```
# Signature in Pandas v0.16
pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
          keys=None, levels=None, names=None, verify_integrity=False, copy=True)
```

`pd.concat()` can be used for a simple concatenation of Series or DataFrame object, just as `np.concatenate()` can be used for simple concatenations of arrays:

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

It also works to concatenate higher-dimensional objects, such as dataframes:

```
df1 = make_df('AB', [1, 2])
df2 = make_df('AB', [3, 4])
display('df1', 'df2', 'pd.concat([df1, df2])')
df1
   A  B
1  A1 B1
2  A2 B2

df2
   A  B
3  A3 B3
4  A4 B4

pd.concat([df1, df2])
   A  B
```

```

1  A1  B1
2  A2  B2
3  A3  B3
4  A4  B4

```

By default, the concatenation takes place row-wise within the dataframe (i.e. `axis=0`). Like `np.concatenate`, `pd.concat` allows specification of an axis along which concatenation will take place. Consider the following example:

```

df3 = make_df('AB', [0, 1])
df4 = make_df('CD', [0, 1])
display('df3', 'df4', "pd.concat([df3, df4], axis='col')")
df3

```

	A	B
0	A0	B0
1	A1	B1

```

df4

```

	C	D
0	C0	D0
1	C1	D1

```

pd.concat([df3, df4], axis='col')

```

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1

We could have equivalently specified `axis=1`; here we've used the more intuitive `axis='col'`.

Duplicate Indices

One important difference between `np.concatenate` and `pd.concat` is that Pandas concatenation **preserves indices**, even if the result will have duplicate indices! Consider this simple example:

```

x = make_df('AB', [0, 1])
y = make_df('AB', [2, 3])
y.index = x.index # make duplicate indices!
display('x', 'y', 'pd.concat([x, y])')
x

```

	A	B
0	A0	B0
1	A1	B1

```

y

```

	A	B
0	A2	B2
1	A3	B3

```

pd.concat([x, y])

```

	A	B
0	A0	B0
1	A1	B1
0	A2	B2
1	A3	B3

```

0  A0  B0
1  A1  B1
0  A2  B2
1  A3  B3

```

Notice the repeated indices in the result. While this is valid within DataFrames, the outcome is often undesirable. `pd.concat()` gives us a few ways to handle it.

Catching the Repeats as an Error. If you'd like to simply verify that the indices in the result of `pd.concat()` do not overlap, you can specify the `verify_integrity` flag. With this set to `True`, the concatenation will raise an exception if there are duplicate indices. Here is an example, where for clarity we'll catch and print the error message:

```

try:
    pd.concat([x, y], verify_integrity=True)
except ValueError as e:
    print("ValueError:", e)
ValueError: Indexes have overlapping values: [0, 1]

```

Ignoring the Index. Sometimes the index itself does not matter, and you would prefer it to simply be ignored. This option can be specified using the `ignore_index` flag. With this set to `True`, the concatenation will create a new integer index for the resulting Series:

```

display('x', 'y', 'pd.concat([x, y], ignore_index=True)')
x
   A  B
0  A0 B0
1  A1 B1

y
   A  B
0  A2 B2
1  A3 B3

pd.concat([x, y], ignore_index=True)
   A  B
0  A0 B0
1  A1 B1
2  A2 B2
3  A3 B3

```

Adding MultiIndex keys. Another option is to use the `keys` option to specify a label for the data sources; the result will be a hierarchically indexed series containing the data:

```

display('x', 'y', "pd.concat([x, y], keys=['x', 'y'])")
x
   A  B
0  A0 B0
1  A1 B1

```

```

y
   A  B
0 A2 B2
1 A3 B3

pd.concat([x, y], keys=['x', 'y'])
   A  B
x 0 A0 B0
  1 A1 B1
y 0 A2 B2
  1 A3 B3

```

The usefulness of this final version should be very apparent; it comes up often when combining data from different sources! The result is a multiply-indexed dataframe, and we can use the tools discussed in Section X.X to transform this data into the representation we're interested in.

Concatenation with Joins

When working with DataFrame objects, the concatenation takes place across one axis, and there are a few options for the set arithmetic to be used on the other columns. Consider the concatenation of the following two dataframes, which have some (but not all!) columns in common:

```

df5 = make_df('ABC', [1, 2])
df6 = make_df('BCD', [3, 4])
display('df5', 'df6', 'pd.concat([df5, df6])')
df5
   A  B  C
1 A1 B1 C1
2 A2 B2 C2

df6
   B  C  D
3 B3 C3 D3
4 B4 C4 D4

pd.concat([df5, df6])
   A  B  C  D
1 A1 B1 C1 NaN
2 A2 B2 C2 NaN
3 NaN B3 C3 D3
4 NaN B4 C4 D4

```

Notice that by default, the entries for which no data is available are filled with NA values. To change this we can specify one of several options for the `join` and `join_axes` parameters of the `concatenate` function. By default, the join is a union of the input columns (`join='outer'`), but we can change this to an intersection of the columns using `join='inner'`:

```
display('df5', 'df6',
        "pd.concat([df5, df6], join='inner')")
```

```
df5
   A  B  C
1 A1 B1 C1
2 A2 B2 C2
```

```
df6
   B  C  D
3 B3 C3 D3
4 B4 C4 D4
```

```
pd.concat([df5, df6], join='inner')
   B  C
1 B1 C1
2 B2 C2
3 B3 C3
4 B4 C4
```

Another option is to directly specify the index of the remaining columns using the `join_axes` argument, which takes a list of index objects. Here we'll specify that the returned columns should be the same as those of the first input:

```
display('df5', 'df6',
        "pd.concat([df5, df6], join_axes=[df5.columns])")
```

```
df5
   A  B  C
1 A1 B1 C1
2 A2 B2 C2
```

```
df6
   B  C  D
3 B3 C3 D3
4 B4 C4 D4
```

```
pd.concat([df5, df6], join_axes=[df5.columns])
   A  B  C
1 A1 B1 C1
2 A2 B2 C2
3 NaN B3 C3
4 NaN B4 C4
```

The combination of options of the `pd.concat` function allows a wide range of possible behaviors when joining two datasets; keep these in mind as you use these tools for your own data!

The `append()` Method

Because direct array concatenation is so common, `Series` and `DataFrame` objects have an `append` method which can accomplish the same thing in fewer keystrokes. For example, rather than calling `pd.concat([df1, df2])`, you can simply call


```
display('df1', 'df2', 'df1.append(df2)')
df1
   A  B
1 A1 B1
2 A2 B2

df2
   A  B
3 A3 B3
4 A4 B4

df1.append(df2)
   A  B
1 A1 B1
2 A2 B2
3 A3 B3
4 A4 B4
```

Keep in mind that unlike the `append()` and `extend()` methods of Python lists, the pandas `append()` method does not modify the original object, but creates a new object with the combined data.

In the next section, we'll look at another more powerful approach to combining data from multiple sources, the database-style merges/joins implemented in `pd.merge`. For more information on `concat()`, `append()`, and related functionality, see the [Merge, Join, and Concatenate](#) section of the Pandas documentation.

Combining Datasets: Merge and Join

One extremely useful feature of Pandas is its high-performance, in-memory join and merge operations. If you have ever worked with databases, you should be familiar with this type of data interaction. The main interface for this is the `pd.merge` function, and we'll see few examples of how this can work in practice.

For convenience, we will start by re-defining the `display()` functionality that we used in the previous notebook:

```
import pandas as pd
import numpy as np

class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style='float: left; padding: 10px;'>
    <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
```

```

        for a in self.args)

def __repr__(self):
    return '\n\n'.join(a + '\n' + repr(eval(a))
                        for a in self.args)

```

Relational Algebra

The behavior implemented in `pd.merge()` is a subset of what is known as *relational algebra*, which is a formal set of rules for manipulating relational data, and forms the conceptual foundation of operations available in most databases. The strength of the relational algebra approach is that it proposes several primitive operations, which become the building blocks of more complicated operations on any dataset. With this lexicon of fundamental operations implemented efficiently in a database or other program, a wide range of fairly complicated composite operations can be performed.

Pandas implements several of these fundamental building-blocks in the `pd.merge()` function and the related `join()` method of Series and Dataframes. As we will see below, these let you efficiently link data from different sources.

Categories of Joins

The `pd.merge()` function implements a number of types of joins, which we'll briefly explore here: the *one-to-one*, *many-to-one*, and *many-to-many* joins. All three types of joins are accessed via an identical call to the `pd.merge()` interface; the type of join performed depends on the form of the input data, as we will see below. Here we will show simple examples of the three types of merges, and discuss detailed options further below.

One-to-One Joins

Perhaps the simplest type of merge expression is the one-to-one join, which is in many ways very similar to the column-wise concatenation seen in Section X.X. As a concrete example, consider the following two dataframes which contain information on several employees in a company:

```

df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering',
                              'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
display('df1', 'df2')
df1

```

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

```
df2
  employee hire_date
0     Lisa      2004
1      Bob      2008
2      Jake      2012
3       Sue      2014
```

To combine this information into a single dataframe, we can use the `pd.merge()` function:

```
display('df1', 'df2', "pd.merge(df1, df2)")
```

```
df1
  employee      group
0      Bob  Accounting
1      Jake  Engineering
2      Lisa  Engineering
3       Sue           HR
```

```
df2
  employee hire_date
0     Lisa      2004
1      Bob      2008
2      Jake      2012
3       Sue      2014
```

```
pd.merge(df1, df2)
  employee      group hire_date
0      Bob  Accounting      2008
1      Jake  Engineering      2012
2      Lisa  Engineering      2004
3       Sue           HR      2014
```

The `pd.merge()` function recognizes that each dataframe has an “employee” column, and automatically joins using this column as a key. The result of the merge is a new dataframe that combines the information from the two inputs. Notice that the order of entries in each column is not necessarily maintained: in the above case, the order of the “employee” column differs between `df1` and `df2`, and the `pd.merge()` function correctly accounts for this. Additionally, keep in mind that the merge in general **discards the index**, except in the special case of merges by index (see the `left_index` and `right_index` keywords, below).

Many-to-One Joins

Many-to-one joins are joins in which one of the two key columns contains duplicate entries. For the many-to-one case, the resulting DataFrame will preserve those duplicate entries as appropriate. Consider the following example of a many-to-one join.

```
df3 = pd.merge(df1, df2)
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
```

```
display('df3', 'df4', "pd.merge(df3, df4)")
df3
  employee    group  hire_date
0      Bob  Accounting    2008
1      Jake Engineering    2012
2      Lisa Engineering    2004
3       Sue         HR    2014

df4
      group supervisor
0  Accounting      Carly
1  Engineering    Guido
2         HR      Steve

pd.merge(df3, df4)
  employee    group  hire_date supervisor
0      Bob  Accounting    2008      Carly
1      Jake Engineering    2012    Guido
2      Lisa Engineering    2004    Guido
3       Sue         HR    2014    Steve
```

The resulting DataFrame has an additional column with the “supervisor” information, where the information is repeated in one or more locations as required by the inputs.

Many-to-Many Joins

Many-to-many joins are a bit confusing conceptually, but are nevertheless well-defined. If the key column in both the left and right array contains duplicates, then the result is a many-to-many merge. This will be perhaps most clear with a concrete example. Consider the following, where we have a DataFrame showing one or more skills associated with a particular group. By performing a many-to-many join, we can recover the skills associated with any individual person:

```
df5 = pd.DataFrame({'group': ['Accounting', 'Accounting', 'Engineering', 'Engineering', 'HR', 'HR'],
                    'skills': ['math', 'spreadsheets', 'coding', 'linux', 'spreadsheets', 'organization']})
display('df1', 'df5', "pd.merge(df1, df5)")
df1
  employee    group
0      Bob  Accounting
1      Jake Engineering
2      Lisa Engineering
3       Sue         HR

df5
      group    skills
0  Accounting    math
1  Accounting spreadsheets
2  Engineering    coding
3  Engineering    linux
```

```

4          HR  spreadsheets
5          HR  organization

pd.merge(df1, df5)
   employee    group    skills
0      Bob  Accounting      math
1      Bob  Accounting  spreadsheets
2      Jake  Engineering      coding
3      Jake  Engineering      linux
4      Lisa  Engineering      coding
5      Lisa  Engineering      linux
6      Sue      HR  spreadsheets
7      Sue      HR  organization

```

These three types of joins can be used with other Pandas tools to implement a wide array of functionality. But in practice, datasets are rarely as clean as shown above. Below we'll consider some of the options provided by `pd.merge()` which enable you to tune how the join operations work.

Specification of the Merge Key

Above we see the default behavior of `pd.merge()`: it looks for one or more matching column names between the two inputs, and uses this as the key. Often the column names will not match so nicely, and `pd.merge()` provides a variety of options for handling this.

The `on` keyword

Most simply, you can explicitly specify the name of the key column using the `on` keyword, which takes a column name or a list of column names:

```

display('df1', 'df2', "pd.merge(df1, df2, on='employee')")
df1
   employee    group
0      Bob  Accounting
1      Jake  Engineering
2      Lisa  Engineering
3      Sue      HR

df2
   employee  hire_date
0      Lisa      2004
1      Bob      2008
2      Jake      2012
3      Sue      2014

pd.merge(df1, df2, on='employee')
   employee    group  hire_date
0      Bob  Accounting      2008
1      Jake  Engineering      2012

```

2	Lisa	Engineering	2004
3	Sue	HR	2014

This option works only if both the left and right DataFrames have the specified column name.

The `left_on` and `right_on` keywords

At times you may wish to merge two datasets with different column names; for example, we may have a dataset in which the employee name is marked by “name” rather than “employee”. In this case, we can use the `left_on` and `right_on` keywords to specify the two column names:

```
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})
display(df1, 'df3', 'pd.merge(df1, df3, left_on="employee", right_on="name")')
df1
```

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

```
df3
```

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

```
pd.merge(df1, df3, left_on="employee", right_on="name")
```

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	90000

The result has a redundant column which we can drop if desired, e.g. using the `drop()` method of DataFrames:

```
pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)
```

	employee	group	salary
0	Bob	Accounting	70000
1	Jake	Engineering	80000
2	Lisa	Engineering	120000
3	Sue	HR	90000

The `left_index` and `right_index` keywords

Sometimes, rather than merging on a column, you would instead like to merge on an index. For example, your data might look like this:

```
df1a = df1.set_index('employee')
df2a = df2.set_index('employee')
display('df1a', 'df2a')
df1a
```

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

```
df2a
```

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

You can use the index as the key for merging by specifying the `left_index` and/or `right_index` flags in `pd.merge()`:

```
display('df1a', 'df2a', "pd.merge(df1a, df2a, left_index=True,
right_index=True)")
df1a
```

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

```
df2a
```

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

```
pd.merge(df1a, df2a, left_index=True, right_index=True)
```

	group	hire_date
employee		
Lisa	Engineering	2004
Bob	Accounting	2008
Jake	Engineering	2012
Sue	HR	2014

For convenience, DataFrames implement the `join()` method, which performs a merge which defaults to joining on indices:

```
display('df1a', 'df2a', 'df1a.join(df2a)')
df1a
```

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

	group	hire_date
employee		
Bob	Accounting	2008
Jake	Engineering	2012
Lisa	Engineering	2004
Sue	HR	2014

If you'd like to mix indices and columns, you can combine `left_index` with `right_on` or `left_on` with `right_index` to get the desired behavior:

```
display('df1a', 'df3', "pd.merge(df1a, df3, left_index=True, right_on='name')")
df1a
```

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

```
pd.merge(df1a, df3, left_index=True, right_on='name')
```

	group	name	salary
0	Accounting	Bob	70000
1	Engineering	Jake	80000
2	Engineering	Lisa	120000
3	HR	Sue	90000

All of these options also work with multiple indices and/or multiple columns; the interface for this behavior is very intuitive. For more information on this see the [Merge, Join, and Concatenate](#) section of the Pandas documentation.

Specifying Set Arithmetic for Joins

In all the above examples we have glossed over one important consideration in performing a join: the type of set arithmetic used in the join. This comes up when a value appears in one key column but not the other; consider this example:

```
df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'food': ['fish', 'lamb', 'bread']},
                   columns=['name', 'food'])
df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                    'drink': ['wine', 'beer']},
                   columns=['name', 'drink'])
display('df6', 'df7', 'pd.merge(df6, df7)')
df6
   name  food
0  Peter  fish
1   Paul  lamb
2   Mary  bread

df7
   name drink
0   Mary  wine
1  Joseph  beer

pd.merge(df6, df7)
   name  food drink
0   Mary  bread  wine
```

Here we have merged two datasets which have only a single “name” entry in common: Mary. By default, the result contains the *intersection* of the two sets of inputs; this is what is known as an *inner join*. We can specify this explicitly using the `how` keyword, which defaults to “inner”:

```
pd.merge(df6, df7, how='inner')
   name  food drink
0   Mary  bread  wine
```

Other options for the `how` keyword are 'outer', 'left', and 'right'. An *outer join* returns a join over the union of the input columns, and fills-in all missing values with NAs

```
display('df6', 'df7', "pd.merge(df6, df7, how='outer')")
df6
   name  food
0  Peter  fish
1   Paul  lamb
2   Mary  bread

df7
   name drink
0   Mary  wine
1  Joseph  beer
```

```
pd.merge(df6, df7, how='outer')
   name  food drink
0  Peter  fish  NaN
1   Paul  lamb  NaN
2   Mary  bread wine
3  Joseph   NaN  beer
```

The *left join* and *right join* return joins over the left entries and right entries, respectively. For example,

```
display('df6', 'df7', "pd.merge(df6, df7, how='left')")
df6
   name  food
0  Peter  fish
1   Paul  lamb
2   Mary  bread

df7
   name drink
0   Mary  wine
1  Joseph  beer

pd.merge(df6, df7, how='left')
   name  food drink
0  Peter  fish  NaN
1   Paul  lamb  NaN
2   Mary  bread wine
```

Notice here that the output rows correspond to the entries in the left input. Using `outer='right'` works in a similar manner.

All of these options can be applied straightforwardly to any of the above join types.

Overlapping Column Names: The `suffixes` Keyword

Finally, you may end up in a case where your two input dataframes have conflicting column names. Consider this example:

```
df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [1, 2, 3, 4]})
df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [3, 1, 4, 2]})
display('df8', 'df9', 'pd.merge(df8, df9, on="name")')
df8
   name  rank
0   Bob    1
1   Jake    2
2   Lisa    3
3   Sue    4

df9
```

	name	rank
0	Bob	3
1	Jake	1
2	Lisa	4
3	Sue	2

```
pd.merge(df8, df9, on="name")
```

	name	rank_x	rank_y
0	Bob	1	3
1	Jake	2	1
2	Lisa	3	4
3	Sue	4	2

Because the output would have two conflicting column names, the merge function automatically appends a suffix "_x" or "_y" to make the output columns unique. If these defaults are inappropriate, it is possible to specify a custom suffix using the `suffixes` keyword:

```
display('df8', 'df9', 'pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])')
```

df8

	name	rank
0	Bob	1
1	Jake	2
2	Lisa	3
3	Sue	4

df9

	name	rank
0	Bob	3
1	Jake	1
2	Lisa	4
3	Sue	2

```
pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])
```

	name	rank_L	rank_R
0	Bob	1	3
1	Jake	2	1
2	Lisa	3	4
3	Sue	4	2

These suffixes work in any of the possible join patterns, and work also if there are multiple overlapping columns.

For more information on these patterns, see Section X.X where we dive a bit deeper into relational algebra. Also see the [Pandas Merge/Join documentation](#) for further discussion of these topics.

Example: US States Data

Merge and Join operations come up most often when combining data from different sources. Here we will consider an example of some data about US states and their populations. The data files can be found at <http://github.com/jakevdp/data-USstates/>

```
# Following are shell commands to download the data
# !curl -O https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-
population.csv
# !curl -O https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-
areas.csv
# !curl -O https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-
abbrevs.csv
```

Let's take a look at the three datasets, using Pandas' `read_csv()` function:

```
pop = pd.read_csv('state-population.csv')
areas = pd.read_csv('state-areas.csv')
abbrevs = pd.read_csv('state-abbrevs.csv')

display('pop.head()', 'areas.head()', 'abbrevs.head()')
pop.head()
  state/region  ages  year  population
0          AL  under18  2012      1117489
1          AL    total  2012      4817528
2          AL  under18  2010      1130966
3          AL    total  2010      4785570
4          AL  under18  2011      1125763

areas.head()
  state  area (sq. mi)
0  Alabama          52423
1  Alaska          656425
2  Arizona          114006
3  Arkansas           53182
4  California         163707

abbrevs.head()
  state abbreviation
0  Alabama          AL
1  Alaska           AK
2  Arizona           AZ
3  Arkansas          AR
4  California        CA
```

Given this information, say we want to compute a relatively straightforward result: **rank US states & territories by their 2010 population density**. We clearly have the data here to find this result, but we'll have to combine the datasets to figure it out.

We'll start with a many-to-one merge which will give us the full state name within the population dataframe. We want to merge based on the "state/region" column of

pop, and the "abbreviation" column of abbrevs. We'll use how='outer' to make sure no data is thrown away due to mis-matched labels.

```
merged = pd.merge(pop, abbrevs, how='outer',
                  left_on='state/region', right_on='abbreviation')
merged = merged.drop('abbreviation', 1) # drop duplicate info
merged.head()
```

	state/region	ages	year	population	state
0	AL	under18	2012	1117489	Alabama
1	AL	total	2012	4817528	Alabama
2	AL	under18	2010	1130966	Alabama
3	AL	total	2010	4785570	Alabama
4	AL	under18	2011	1125763	Alabama

Let's double-check whether there was any mis-matches here; we can do this by checking for rows with nulls:

```
merged.isnull().any()
state/region    False
ages            False
year            False
population       True
state           True
dtype: bool
```

Some of the "population" info is Null; let's figure out which these are!

```
merged[merged['population'].isnull()].head()
```

	state/region	ages	year	population	state
2448	PR	under18	1990	NaN	NaN
2449	PR	total	1990	NaN	NaN
2450	PR	total	1991	NaN	NaN
2451	PR	under18	1991	NaN	NaN
2452	PR	total	1993	NaN	NaN

It appears that all the null population values are from Puerto Rico prior to the year 2000; this is likely due to this data not being available from the original source.

More importantly, we see also that some of the new "state" entries are also Null, which means that there was no corresponding entry in the abbrevs key! Let's figure out which regions lack this match:

```
merged['state/region'][merged['state'].isnull()].unique()
array(['PR', 'USA'], dtype=object)
```

We can quickly infer the issue: our population data includes entries for Puerto Rico (PR) and the United States as a whole (USA), while these entries do not appear in the state abbreviation key. We can fix these quickly by filling-in appropriate entries:

```
merged.loc[merged['state/region'] == 'PR', 'state'] = 'Puerto Rico'
merged.loc[merged['state/region'] == 'USA', 'state'] = 'United States'
merged.isnull().any()
state/region    False
```

```

ages           False
year           False
population      True
state          False
dtype: bool

```

No more NULLs in the state column: we're all set!

Now we can merge the result with the area data using a similar procedure. Examining what's above, we will want to join on the 'state' column in both:

```

final = pd.merge(merged, areas, on='state', how='left')
final.head()

```

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489	Alabama	52423
1	AL	total	2012	4817528	Alabama	52423
2	AL	under18	2010	1130966	Alabama	52423
3	AL	total	2010	4785570	Alabama	52423
4	AL	under18	2011	1125763	Alabama	52423

Again, let's check for Nulls to see if there were any mis-matches.

```

final.isnull().any()
state/region      False
ages              False
year              False
population         True
state             False
area (sq. mi)     True
dtype: bool

```

There are NULLs in the area column; we can take a look to see which regions were ignored here:

```

final['state'][final['area (sq. mi)'].isnull()].unique()
array(['United States'], dtype=object)

```

We see that our areas DataFrame does not contain the area of the United States as a whole. We could insert the appropriate value (using, e.g. the sum of all state areas), but in this case we'll just drop the null values because the population density of the entire US is not relevant to our current discussion:

```

final.dropna(inplace=True)
final.head()

```

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489	Alabama	52423
1	AL	total	2012	4817528	Alabama	52423
2	AL	under18	2010	1130966	Alabama	52423
3	AL	total	2010	4785570	Alabama	52423
4	AL	under18	2011	1125763	Alabama	52423

Now we have all the data we need. To answer the question of interest, let's first select the portion of the data corresponding with the year 2000, and the total population. We'll use the `query()` function to do this quickly (see Section X.X):

```
data2010 = final.query("year == 2010 & ages == 'total'")
data2010.head()
  state/region  ages  year  population  state  area (sq. mi)
3           AL  total  2010     4785570  Alabama      52423
91          AK  total  2010     713868  Alaska      656425
101         AZ  total  2010     6408790  Arizona      114006
189         AR  total  2010     2922280  Arkansas      53182
197         CA  total  2010    37333601  California     163707
```

Now let's compute the population density and display it in order. We'll start by re-indexing our data on the state, and then compute the result.

```
data2010.set_index('state', inplace=True)
density = data2010['population'] / data2010['area (sq. mi)']

density.sort(ascending=False)
density.head()
state
District of Columbia    8898.897059
Puerto Rico            1058.665149
New Jersey              1009.253268
Rhode Island            681.339159
Connecticut             645.600649
dtype: float64
```

The result is a ranking of US states plus Washington DC and Puerto Rico in order of their population density, in residents per square mile. We can see that Washington DC is by far the densest region in this dataset; the densest US state is New Jersey.

We can also check the end of the list:

```
density.tail()
state
South Dakota    10.583512
North Dakota    9.537565
Montana         6.736171
Wyoming         5.768079
Alaska          1.087509
dtype: float64
```

We see that the least dense state, by far, is Alaska, with just over one person per square mile.

This type of messy data merging is a common task when trying to answer questions based on real-world data. I hope that this example has given you an idea of the ways you can combine tools we've learned about to gain insight from real-world data!

Aggregation and Grouping

An essential piece of analysis of large data is efficient summarization: computing aggregations like `sum()`, `mean()`, `median()`, `min()`, and `max()`, in which a single number gives insight into the nature of a potentially large dataset. In this section we'll explore aggregations in Pandas, from simple operations akin to what we've seen on NumPy arrays, to more sophisticated operations based on the concept of a *Group-By*. We'll see examples of these below.

For convenience, we'll use the same display magic function that we've seen in previous sections:

```
import numpy as np
import pandas as pd

class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style='float: left; padding: 10px;'>
<p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
</div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                          for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                            for a in self.args)
```

Planets Data

Below we will use the *planets* dataset, available via the [seaborn](#) library. It gives information on planets that astronomers have discovered around other stars (known as *extrasolar planets* or *exoplanets* for short). It can be downloaded with a simple `seaborn` command:

```
import seaborn as sns
planets = sns.load_dataset('planets')
planets.shape
(1035, 6)

planets.head()
```

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009

This has some details on the more than thousand planets discovered around other stars up to 2014.

Simple Aggregation in Pandas

In Section X.X, we explored some of the data aggregations available for NumPy arrays. For a Pandas Series, similarly to a one-dimensional NumPy array, the aggregates return a single value:

```
rng = np.random.RandomState(42)
ser = pd.Series(rng.rand(5))
ser
0    0.374540
1    0.950714
2    0.731994
3    0.598658
4    0.156019
dtype: float64

ser.sum()
2.8119254917081569

ser.mean()
0.56238509834163142
```

For a DataFrame, by default the aggregates return results within each column:

```
df = pd.DataFrame({'A': rng.rand(5),
                   'B': rng.rand(5)})
df
   A      B
0  0.155995 0.020584
1  0.058084 0.969910
2  0.866176 0.832443
3  0.601115 0.212339
4  0.708073 0.181825

df.mean()
A    0.477888
B    0.443420
dtype: float64
```

By specifying the `axis` argument, you can instead aggregate along the columns instead:

```
df.mean(axis=1)
0    0.088290
1    0.513997
2    0.849309
3    0.406727
4    0.444949
dtype: float64
```

Pandas Series and DataFrames include all of the common aggregates mentioned in Section X.X; in addition there is a convenience method `describe()` which computes several common aggregates for each column and returns the result. Let's use this on the planets data:

```
planets.describe()
```

	number	orbital_period	mass	distance	year
count	1035.000000	992.000000	513.000000	808.000000	1035.000000
mean	1.785507	2002.917596	2.638161	264.069282	2009.070531
std	1.240976	26014.728304	3.818617	733.116493	3.972567
min	1.000000	0.090706	0.003600	1.350000	1989.000000
25%	1.000000	5.442540	0.229000	32.560000	2007.000000
50%	1.000000	39.979500	1.260000	55.250000	2010.000000
75%	2.000000	526.005000	3.040000	178.500000	2012.000000
max	7.000000	730000.000000	25.000000	8500.000000	2014.000000

This can be a useful way to begin to understand a dataset. For example, we see in the `year` column that although exoplanets have been discovered since 1989, half of all known exoplanets were discovered since 2010! This is largely thanks to the *Kepler* mission, which is a space-based telescope specifically designed for finding eclipsing planets around other stars.

Other built-in aggregations in Pandas are summarized in the following table:

Aggregation	Description
<code>count()</code>	total number of items
<code>first(), last()</code>	first or last item
<code>mean(), median()</code>	mean or median
<code>min(), max()</code>	minimum or maximum
<code>std(), var()</code>	standard deviation or variance
<code>mad()</code>	mean absolute deviation
<code>prod()</code>	product of all items
<code>sum()</code>	sum of all items

These are all methods of DataFrame and Series objects.

To go deeper into the data, however, simple aggregates are often not enough. The next level of data summarization is the *Group By* operation, which allows you to quickly and efficiently compute aggregates on subsets of data.

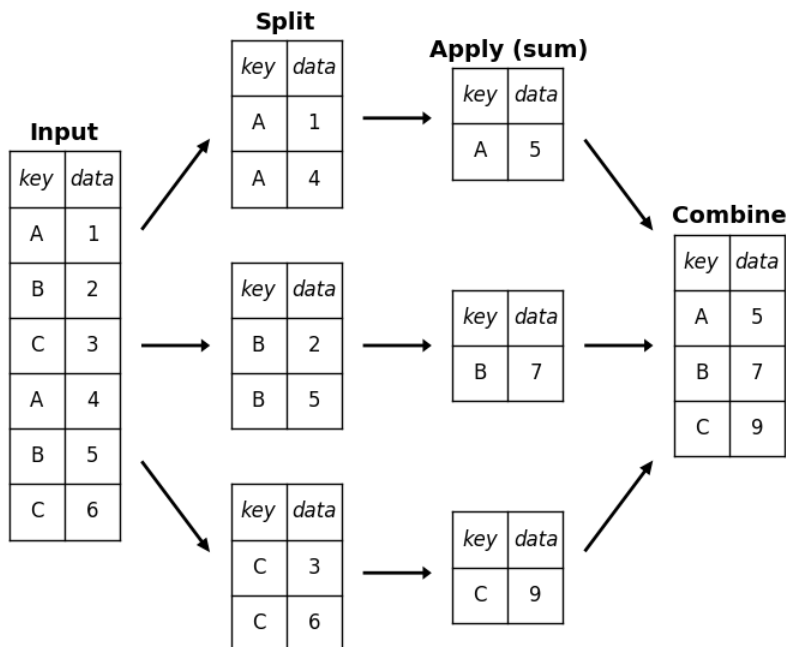
Group By: Split, Apply, Combine

Simple aggregations can give you a flavor of your dataset, but often we would prefer to aggregate conditionally on some label or index: this is implemented in the so-called *Group By* operation. The name “Group By” comes from a command in the SQL

database language, but it is perhaps more illuminative to think of it in the terms first coined by Hadley Wickham of Rstats fame: *Split, Apply, Combine*.

Split, Apply, Combine

A canonical example of this split-apply-combine operation, where the “apply” is a summation aggregation, is illustrated below:



This figure makes abundantly clear what the `GroupBy` accomplishes:

- The **split** step involves breaking up and grouping a dataframe depending on the value of a particular key.
- The **apply** step involves computing some function, usually an aggregate, transformation, or filtering, over individual groups.
- The **combine** step merges the results of these operations into an output array.

While this could certainly be done manually using some combination of the masking, aggregation, and merging commands covered earlier, an important realization is that *the intermediate splits do not need to be explicitly instantiated*. It is quite easy to imagine creating a streaming algorithm which would simply update the sum, mean, count, min, or other aggregate for each group during a single pass over the data. The power of the `GroupBy` is that it abstracts-away these steps: a simple interface to this func-

tionality wraps the more sophisticated computational decisions taking place under the hood.

As a concrete example, let's take a look at using Pandas for the above computation. We'll start by creating the input DataFrame:

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(6)}, columns=['key', 'data'])
```

	key	data
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

The most basic split-apply-combine operation can be computed with the `groupby()` method of dataframes, passing the name of the desired key column:

```
df.groupby('key')
<pandas.core.groupby.DataFrameGroupBy object at 0x1024503d0>
```

Notice that what is returned is not a set of DataFrames, but a `DataFrameGroupBy` object. This object is where the magic is: you can think of it as a special view of the DataFrame, which does no computation until the aggregation is applied. This lazy evaluation approach means that common aggregates can be implemented very efficiently in a way that is almost transparent to the user.

To produce a result, we can apply an aggregate to this `DataFrameGroupBy` object, which will perform the appropriate apply/combine steps to produce the desired result:

```
df.groupby('key').sum()
```

	data
key	
A	3
B	5
C	7

Notice that the *apply-combine* is accomplished within a single step. The `sum()` method is just one possibility here: you can apply virtually any common Pandas or NumPy aggregation function, as well as virtually any valid DataFrame operation, as we will see below.

The GroupBy Object

The GroupBy object is a very flexible abstraction. In many ways, you can simply treat it as if it's a collection of DataFrames, and it does the difficult things under the hood.

Here are some examples, using the planets again:

Aggregate, Filter, Transform, Apply. The native operations built on GroupBy can be broadly split into *aggregation*, which we saw above, *filter*, which selects groups based on some criterion, *transform* which modifies groups more flexibly than aggregation, and *apply*, which is a general umbrella over operations on a group.

Because these are all very important and involved, we will discuss them in their own section further below.

Column Indexing. The GroupBy object supports column indexing in the same way as the DataFrame, and returns a modified GroupBy object. For example:

```
planets.groupby('method')
<pandas.core.groupby.DataFrameGroupBy object at 0x102450910>

planets.groupby('method')['orbital_period']
<pandas.core.groupby.SeriesGroupBy object at 0x102450f10>
```

Here we've selected a particular Series group from the original DataFrame group by reference to its column name. We can then compute any aggregate or other operation on this sub-GroupBy:

```
planets.groupby('method')['orbital_period'].median()
method
Astrometry                631.180000
Eclipse Timing Variations  4343.500000
Imaging                    27500.000000
Microlensing               3300.000000
Orbital Brightness Modulation  0.342887
Pulsar Timing              66.541900
Pulsation Timing Variations 1170.000000
Radial Velocity            360.200000
Transit                    5.714932
Transit Timing Variations  57.011000
Name: orbital_period, dtype: float64
```

This gives an idea of the general scale of orbital periods (in days) that the methods are sensitive to.

Iteration over groups. The GroupBy object supports direct iteration over the groups, returning each group as a Series or DataFrame:

```
for (method, group) in planets.groupby('method'):
    print("{0:30s} shape={1}".format(method, group.shape))
Astrometry                shape=(2, 6)
Eclipse Timing Variations shape=(9, 6)
Imaging                    shape=(38, 6)
Microlensing               shape=(23, 6)
Orbital Brightness Modulation shape=(3, 6)
Pulsar Timing              shape=(5, 6)
Pulsation Timing Variations shape=(1, 6)
Radial Velocity            shape=(553, 6)
```

```
Transit                shape=(397, 6)
Transit Timing Variations  shape=(4, 6)
```

This can be useful for doing certain things manually, though it is often much faster to use the built-in GroupBy functionality, discussed further below.

Dispatch Methods. Through some Python class magic, any method not explicitly implemented by the GroupBy object will be passed through and called on the groups, whether they are DataFrame or Series objects. For example, you can use the describe() method of DataFrames to perform a set of aggregations which describe each group in the data.

```
planets.groupby('method')['year'].describe().unstack()
```

	count	mean	std	min	25%	\
method						
Astrometry	2	2011.500000	2.121320	2010	2010.75	
Eclipse Timing Variations	9	2010.000000	1.414214	2008	2009.00	
Imaging	38	2009.131579	2.781901	2004	2008.00	
Microlensing	23	2009.782609	2.859697	2004	2008.00	
Orbital Brightness Modulation	3	2011.666667	1.154701	2011	2011.00	
Pulsar Timing	5	1998.400000	8.384510	1992	1992.00	
Pulsation Timing Variations	1	2007.000000	NaN	2007	2007.00	
Radial Velocity	553	2007.518987	4.249052	1989	2005.00	
Transit	397	2011.236776	2.077867	2002	2010.00	
Transit Timing Variations	4	2012.500000	1.290994	2011	2011.75	

	50%	75%	max
method			
Astrometry	2011.5	2012.25	2013
Eclipse Timing Variations	2010.0	2011.00	2012
Imaging	2009.0	2011.00	2013
Microlensing	2010.0	2012.00	2013
Orbital Brightness Modulation	2011.0	2012.00	2013
Pulsar Timing	1994.0	2003.00	2011
Pulsation Timing Variations	2007.0	2007.00	2007
Radial Velocity	2009.0	2011.00	2014
Transit	2012.0	2013.00	2014
Transit Timing Variations	2012.5	2013.25	2014

Looking at this table helps us to better understand the data: for example, the vast majority of planets have been discovered by the **Radial Velocity** and **Transit** methods, though the latter only became common (due to new, more accurate telescopes) in the last decade. The newest methods seem to be **Transit Timing Variation** and **Orbital Brightness Modulation**, which were not used to discover a new planet until 2011.

This is just one example of the utility of dispatch methods. Notice that they are applied to *each DataFrame/Series in the group*, and the results are then combined within GroupBy and returned. Again, any valid DataFrame/Series method can be

used on the corresponding GroupBy object, which allows for some very flexible and powerful operations!

Aggregate, Filter, Transform, Apply

Above we have been focusing on aggregation for the combine operation, but there are more options available. In particular, GroupBy objects have an `aggregate()`, `filter()`, `transform()`, and `apply()` methods which efficiently implement a variety of useful operations before combining the grouped data.

For the purpose of the selections below, we'll use the following dataframe:

```
rng = np.random.RandomState(0)
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                  'data1': range(6),
                  'data2': rng.randint(0, 10, 6)},
                  columns = ['key', 'data1', 'data2'])
```

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

Aggregation. We're now familiar with GroupBy aggregations, but the `aggregate()` method allows for even more flexibility. It can take a string, a function, or a list thereof. Here is a quick example combining all these:

```
df.groupby('key').aggregate(['min', np.median, max])
```

	data1			data2		
	min	median	max	min	median	max
key						
A	0	1.5	3	3	4.0	5
B	1	2.5	4	0	3.5	7
C	2	3.5	5	3	6.0	9

Another useful pattern is to pass a dictionary mapping column names to aggregation operations

```
df.groupby('key').aggregate({'data1': 'min',
                             'data2': 'max'})
```

	data2	data1
key		
A	5	0
B	7	1
C	9	2

Filtering. A filtering operation allows you to drop data based on the group properties. For example, we might want to keep all groups in which the standard deviation meets some cut:

```
def filter_func(x):
    return x['data2'].std() > 4

display('df', "df.groupby('key').std()", "df.groupby('key').filter(fil-
ter_func)")
df
```

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

```
df.groupby('key').std()
      data1      data2
key
A    2.12132  1.414214
B    2.12132  4.949747
C    2.12132  4.242641

df.groupby('key').filter(filter_func)
      key  data1  data2
1    B      1      0
2    C      2      3
4    B      4      7
5    C      5      9
```

The filter function should return a boolean value specifying whether the group passes the filtering. Here because group A does not have a standard deviation greater than 4, it is dropped from the filtering operation.

Transformation. While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine. A common example is to center the data by subtracting the group-wise mean:

```
center = lambda x: x - x.mean()
df.groupby('key').transform(center)
```

	data1	data2
0	-1.5	1.0
1	-1.5	-3.5
2	-1.5	-3.0
3	1.5	-1.0
4	1.5	3.5
5	1.5	3.0

Apply Method. The `apply()` method lets you apply an arbitrary function to the group results. The function should take a dataframe, and return either a Pandas object (e.g. DataFrame, Series) or a scalar; the combine operation will be tailored to the type of output returned.

For example, here is an `apply()` which normalizes the first column by the sum of the second:

```
def norm_by_data2(x):
    # x is a DataFrame of group values
    x['data1'] /= x['data2'].sum()
    return x

display('df', "df.groupby('key').apply(norm_by_data2)")
df
```

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

```
df.groupby('key').apply(norm_by_data2)
```

	key	data1	data2
0	A	0.000000	5
1	B	0.142857	0
2	C	0.166667	3
3	A	0.375000	3
4	B	0.571429	7
5	C	0.416667	9

`apply()` within a GroupBy is extremely flexible: the only criterion is that the function takes a dataframe and returns a Pandas object or scalar; what you do in the middle is up to you!

Specifying Groups

In the above simple examples, we have split the DataFrame on a single column name. This is just one of many options by which the groups can be defined, and we'll go through some other options for group specification here:

A list, array, series, or index providing the grouping keys. The key can be any appropriate series. For example:

```
L = [0, 1, 0, 1, 2, 0]
display('df', 'df.groupby(L).sum()')
df
```

	key	data1	data2
0	A	0	5

1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

```
df.groupby(L).sum()
      data1  data2
0         7     17
1         4         3
2         4         7
```

Of course, this means there's another, more verbose way of accomplishing the `df.groupby('key')` from above:

```
display('df', "df.groupby(df['key']).sum()")
df
      key  data1  data2
0     A         0         5
1     B         1         0
2     C         2         3
3     A         3         3
4     B         4         7
5     C         5         9
```

```
df.groupby(df['key']).sum()
      data1  data2
key
A         3         8
B         5         7
C         7        12
```

A dictionary or series mapping index to group. Another method is to provide a dictionary which maps index values to the group keys:

```
df2 = df.set_index('key')
mapping = {'A': 'vowel', 'B': 'consonant', 'C': 'consonant'}
display('df2', 'df2.groupby(mapping).sum()')
df2
      data1  data2
key
A         0         5
B         1         0
C         2         3
A         3         3
B         4         7
C         5         9

df2.groupby(mapping).sum()
      data1  data2
consonant    12    19
vowel         3         8
```

Any Python function. Similarly to the mapping, you can pass any Python function which will input the index value and output the group:

```
display('df2', 'df2.groupby(str.lower).mean()')
```

```
df2
      data1  data2
key
A         0      5
B         1      0
C         2      3
A         3      3
B         4      7
C         5      9
```

```
df2.groupby(str.lower).mean()
```

```
      data1  data2
a      1.5    4.0
b      2.5    3.5
c      3.5    6.0
```

A list of valid keys. Further, any of the above key choices can be combined to group on a multi-index:

```
df2.groupby([str.lower, mapping]).mean()
```

```
      data1  data2
a vowel      1.5    4.0
b consonant  2.5    3.5
c consonant  3.5    6.0
```

Grouping Example. As an example of this, in a couple lines of Python code we can put all these together and count discovered planets by method and by decade:

```
decade = 10 * (planets['year'] // 10)
decade = decade.astype(str) + 's'
decade.name = 'decade'
planets.groupby(['method', decade])['number'].sum().unstack().fillna(0)
```

```
decade
method      1980s  1990s  2000s  2010s
Astrometry           0     0     0     2
Eclipse Timing Variations  0     0     5    10
Imaging              0     0    29    21
Microlensing         0     0    12    15
Orbital Brightness Modulation  0     0     0     5
Pulsar Timing        0     9     1     1
Pulsation Timing Variations  0     0     1     0
Radial Velocity       1    52   475   424
Transit              0     0    64   712
Transit Timing Variations  0     0     0     9
```

This shows the power of combining many of the operations we've discussed up to this point when looking at realistic datasets. We immediately gain a coarse understanding of when and how planets have been discovered over the past several decades!

Here I would suggest digging into the above few lines of code, and evaluating the individual steps to make sure you understand exactly what they are doing to the result. It's certainly a rather complicated example, but once you understand the pieces you'll understand the process!

Pivot Tables

In the previous section, we looked at grouping operations. A *pivot table* is a related operation which is commonly seen in spreadsheets and other programs which operate on tabular data. The Pivot Table takes simple column-wise data as input, and groups the entries into a two-dimensional table which provides a multi-dimensional summarization of the data. The difference between Pivot Tables and GroupBy can sometimes cause confusion; it helps me to think of pivot tables as essentially a **multi-dimensional** version of GroupBy aggregation. That is, you split-apply-combine, but both the split and the combine happen across not a one-dimensional index, but across a two-dimensional grid.

Motivating Pivot Tables

For the examples in this section, we'll use the database of passengers on the Titanic, available through the seaborn library:

```
import numpy as np
import pandas as pd
import seaborn as sns
titanic = sns.load_dataset('titanic')

titanic.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	\
0	0	3	male	22	1	0	7.2500	S	Third	
1	1	1	female	38	1	0	71.2833	C	First	
2	1	3	female	26	0	0	7.9250	S	Third	
3	1	1	female	35	1	0	53.1000	S	First	
4	0	3	male	35	0	0	8.0500	S	Third	

	who	adult_male	deck	embark_town	alive	alone
0	man	True	NaN	Southampton	no	False
1	woman	False	C	Cherbourg	yes	False
2	woman	False	NaN	Southampton	yes	True
3	woman	False	C	Southampton	yes	False
4	man	True	NaN	Southampton	no	True

This contains a wealth of information on each passenger of that ill-fated voyage, including their gender, age, class, fare paid, and much more.

Pivot Tables By Hand

To start learning more about this data, we might want to like to group it by gender, survival, or some combination thereof. If you have read the previous section, you might be tempted to apply a GroupBy operation to this data. For example, let's look at survival rate by gender:

```
titanic.groupby('sex')[['survived']].mean()
survived
sex
female  0.742038
male    0.188908
```

This immediately gives us some insight: overall three of every four females on board survived, while only one in five males survived!

This is an interesting insight, but we might like to go one step deeper and look at survival by both sex and, say, class. Using the vocabulary of GroupBy, we might proceed something like this: We *group by* class and gender, *select* survival, *apply* a mean aggregate, *combine* the resulting groups, and then *unstack* the hierarchical index to reveal the hidden multidimensionality. In code:

```
titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').unstack()
class      First      Second      Third
sex
female  0.968085  0.921053  0.500000
male    0.368852  0.157407  0.135447
```

This gives us a better idea of how both gender and class affected survival, but the code is starting to look a bit garbled. While each step of this pipeline makes sense in light of the tools we've previously discussed, the long string of code is not particularly easy to read or use. This type of operation is common enough that Pandas includes a convenience routine, `pivot_table`, which succinctly handles this type of multidimensional aggregation.

Pivot Table Syntax

Here is the equivalent to the above operation using the `pivot_table` method of dataframes:

```
titanic.pivot_table('survived', index='sex', columns='class')
class      First      Second      Third
sex
female  0.968085  0.921053  0.500000
male    0.368852  0.157407  0.135447
```

This is eminently more readable than the equivalent GroupBy operation, and produces the same result. As you might expect of an early 20th century transatlantic cruise, the survival gradient favors both women and higher classes. First-class women

survived with near certainty (hi Kate!), while only one in ten third-class men survived (sorry Leo!).

Multi-level Pivot Tables

Just as in the GroupBy, the grouping in pivot tables can be specified with multiple levels, and via a number of options. For example, we might be interested in looking at age as a third dimension. We'll bin the age using the `pd.cut` function:

```
age = pd.cut(titanic['age'], [0, 18, 80])
titanic.pivot_table('survived', ['sex', age], 'class')
```

		First	Second	Third
sex	age			
female	(0, 18]	0.909091	1.000000	0.511628
	(18, 80]	0.972973	0.900000	0.423729
male	(0, 18]	0.800000	0.600000	0.215686
	(18, 80]	0.375000	0.071429	0.133663

we can do the same game with the columns; let's add info on the fare paid using `pd.qcut` to automatically compute quantiles:

```
fare = pd.qcut(titanic['fare'], 2)
titanic.pivot_table('survived', ['sex', age], [fare, 'class'])
```

		[0, 14.454]	(14.454, 512.329]	\	
		First	Second	Third	
sex	age				
female	(0, 18]	NaN	1.000000	0.714286	0.909091 1.000000
	(18, 80]	NaN	0.880000	0.444444	0.972973 0.914286
male	(0, 18]	NaN	0.000000	0.260870	0.800000 0.818182
	(18, 80]	0	0.098039	0.125000	0.391304 0.030303

		Third
fare		
class		
sex	age	
female	(0, 18]	0.318182
	(18, 80]	0.391304
male	(0, 18]	0.178571
	(18, 80]	0.192308

The result is a four-dimensional aggregation, shown in a grid which demonstrates the relationship between the values.

Additional Pivot Table Options

The full call signature of the `pivot_table` method of DataFrames is as follows:

```
DataFrame.pivot_table(values=None, index=None, columns=None, aggfunc='mean',
                        fill_value=None, margins=False, dropna=True)
```

Above we've seen examples of the first three arguments; here we'll take a quick look at the remaining arguments. Two of the options, `fill_value` and `dropna`, have to do

with missing data and are fairly straightforward; we will not show examples of them here.

The `aggfunc` keyword controls what type of aggregation is applied, which is a mean by default. As in the `GroupBy`, the aggregation specification can be a string representing one of several common choices (e.g. 'sum', 'mean', 'count', 'min', 'max', etc.) or a function which implements an aggregation (e.g. `np.sum()`, `min()`, `sum()`, etc.). Additionally, it can be specified as dictionary mapping a column to any of the above desired options:

```
titanic.pivot_table(index='sex', columns='class',
                    aggfunc={'survived':sum, 'fare':'mean'})
```

	survived			fare		
class	First	Second	Third	First	Second	Third
sex						
female	91	70	72	106.125798	21.970121	16.118810
male	45	17	47	67.226127	19.741782	12.661633

Notice also here that we've omitted the `values` keyword; when specifying a mapping for `aggfunc`, this is determined automatically.

At times it's useful to compute totals along each grouping. This can be done via the `margins` keyword:

```
titanic.pivot_table('survived', index='sex', columns='class', margins=True)
```

class	First	Second	Third	All
sex				
female	0.968085	0.921053	0.500000	0.742038
male	0.368852	0.157407	0.135447	0.188908
All	0.629630	0.472826	0.242363	0.383838

Here this automatically gives us information about the class-agnostic survival rate by gender, the gender-agnostic survival rate by class, and the overall survival rate of 38%.

Example: Birthrate Data

As a more interesting example, let's take a look at the freely-available data on births in the USA, provided by the Centers for Disease Control (CDC). This data can be found at <https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv>. This dataset has been analyzed rather extensively by Andrew Gelman and his group; see for example [this blog post](#).

```
# shell command to download the data:
# !curl -O https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/
births.csv

births = pd.read_csv('births.csv')
```

Taking a look at the data, we see that it's relatively simple: it contains the number of births grouped by date and gender:

```
births.head()
   year  month  day  gender  births
0  1969     1    1      F    4046
1  1969     1    1      M    4440
2  1969     1    2      F    4454
3  1969     1    2      M    4548
4  1969     1    3      F    4548
```

We can start to understand this data a bit more by using a pivot table. Let's add a decade column, and take a look at male and female births as a function of decade:

```
births['decade'] = 10 * (births['year'] // 10)
births.pivot_table('births', index='decade', columns='gender', aggfunc='sum')
gender      F      M
decade
1960      1753634  1846572
1970      16263075  17121550
1980      18310351  19243452
1990      19479454  20420553
2000      18229309  19106428
```

We immediately see that male births outnumber female births in every decade. To see this trend a bit more clearly, we can use Pandas' built-in plotting tools to visualize the total number of births by year (see Chapter X.X for a discussion of plotting with matplotlib):

```
%matplotlib inline
import matplotlib.pyplot as plt
sns.set() # use seaborn styles
births.pivot_table('births', index='year', columns='gender', agg-
func='sum').plot()
plt.ylabel('total births per year');
```

With a simple pivot table and plot() method, we can immediately see the annual trend in births by gender. By eye, we find that over the past 50 years male births have outnumbered female births by around 5%.

Further Data Exploration

Though this doesn't necessarily relate to the pivot table, there are a few more interesting features we can pull out of this dataset using the Pandas tools covered up to this point. We must start by cleaning the data a bit, removing outliers caused by mistyped dates (e.g. June 31st) or missing values (e.g. June 99th). One easy way to remove these all at once is to cut outliers; we'll do this via a robust sigma-clipping operation:

```
# Some data is mis-reported; e.g. June 31st, etc.
# remove these outliers via robust sigma-clipping
quartiles = np.percentile(births['births'], [25, 50, 75])
mu = quartiles[1]
sig = 0.7413 * (quartiles[2] - quartiles[0])
births = births.query('(births > @mu - 5 * @sig) & (births < @mu + 5 * @sig)')
```


Next we set the day column to integers; previously it had been a string because some columns in the dataset contained the value 'null':

```
# set 'day' column to integer; it originally was a string due to nulls
births['day'] = births['day'].astype(int)
```

Finally, we can combine the day, month, and year to create a Date index (see section X.X). This allows us to quickly compute the weekday corresponding to each row:

```
# create a datetime index from the year, month, day
births.index = pd.to_datetime(10000 * births.year +
                              100 * births.month +
                              births.day, format='%Y%m%d')
```

```
births['dayofweek'] = births.index.dayofweek
```

Using this we can plot births by weekday for several decades:

```
import matplotlib.pyplot as plt
import matplotlib as mpl

births.pivot_table('births', index='dayofweek',
                   columns='decade', aggfunc='mean').plot()
plt.gca().set_xticklabels(['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun'])
plt.ylabel('mean births by day');
```

Apparently births are slightly less common on weekends than on weekdays! Note that the 1990s and 2000s are missing because the CDC stopped reports only the month of birth starting in 1989.

Another interesting view is to plot the mean number of births by the day of the *year*. We can do this by constructing a datetime array for a particular year, making sure to choose a leap year so as to account for February 29th.

```
# Choose a leap year to display births by date
dates = [pd.datetime(2012, month, day)
         for (month, day) in zip(births['month'], births['day'])]
```

We can now group by the data by day of year and plot the results. We'll additionally annotate the plot with the location of several US holidays:

```
# Plot the results
fig, ax = plt.subplots(figsize=(8, 6))
births.pivot_table('births', dates).plot(ax=ax)

# Label the plot
ax.text('2012-1-1', 3950, "New Year's Day")
ax.text('2012-7-4', 4250, "Independence Day", ha='center')
ax.text('2012-9-4', 4850, "Labor Day", ha='center')
ax.text('2012-10-31', 4600, "Halloween", ha='right')
ax.text('2012-11-25', 4450, "Thanksgiving", ha='center')
ax.text('2012-12-25', 3800, "Christmas", ha='right')
ax.set(title='USA births by day of year (1969-1988)',
```

```

ylabel='average daily births',
xlim=('2011-12-20','2013-1-10'),
ylim=(3700, 5400));

# Format the x axis with centered month labels
ax.xaxis.set_major_locator(mpl.dates.MonthLocator())
ax.xaxis.set_minor_locator(mpl.dates.MonthLocator(bymonthday=15))
ax.xaxis.set_major_formatter(plt.NullFormatter())
ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%h'));

```

The lower birthrate on holidays is striking, but is likely the result of selection for scheduled/induced births rather than any deep psychosomatic causes. For more discussion on this trend, see the discussion and links in [Andrew Gelman's blog posts](#) on the subject.

This short example should give you a good idea of how many of the Pandas tools we've seen to this point can be put together and used to gain insight from a variety of datasets. We will see some more sophisticated analysis of this data, and other datasets like it, in future sections!

Vectorized String Operations

Python has a very nice set of built-in operations for manipulating strings; we covered some of the basics in Section X.X. Pandas builds on this and provides a comprehensive set of *vectorized string operations* which become an essential piece of the type of munging required when working with real-world data.

In this section, we'll walk through some of the Pandas string operations, and then take a look at using them to partially clean-up a very messy dataset of recipes collected from the internet.

Introducing Pandas String Operations

We saw in previous sections how tools like NumPy and Pandas generalize arithmetic operations so that we can easily and quickly perform the same operation on many array elements; for example:

```

import numpy as np
x = np.array([2, 3, 5, 7, 11, 13])
x * 2
array([ 4,  6, 10, 14, 22, 26])

```

This *vectorization* of operations simplifies the syntax of operating on arrays of data: we no longer have to worry about the size or shape of the array, but just about what operation we want done.

For arrays of strings, NumPy does not provide such simple access, and thus you're stuck using a more verbose loop syntax

```
data = ['peter', 'Paul', 'MARY', 'gUIDO']
[s.capitalize() for s in data]
['Peter', 'Paul', 'Mary', 'Guido']
```

This is perhaps sufficient to work with data, but it will break if there are any missing values. For example:

```
data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
[s.capitalize() for s in data]
```

Pandas includes features to address both this need for vectorized string operations and for correctly handling missing data through a bit of Python attribute magic. Pandas does this using the `str` attribute of Pandas Series, DataFrames, and Index objects. So, for example, if we create a Pandas Series with this data,

```
import pandas as pd
names = pd.Series(data)
names
0    peter
1     Paul
2     None
3     MARY
4    gUIDO
dtype: object
```

We can now call a single method which will capitalize all the entries, while skipping over any missing values:

```
names.str.capitalize()
0    Peter
1     Paul
2     None
3     Mary
4     Guido
dtype: object
```

This `str` attribute of Pandas objects is a special attribute which contains all the vectorized string methods available to Pandas.

Tables of Pandas String Methods

If you have a good understanding of string manipulation in Python, most of Pandas string syntax is intuitive enough that it's probably sufficient to just list a table of available methods; we will start with that here, before diving deeper into a few of the subtleties.

Methods Similar to Python String Methods

Nearly all Python string methods have an equivalent Pandas vectorized string method; see Section X.X where a few of these are discussed.

Method	Description	Method	Description
<code>len()</code>	Equivalent to <code>len(str)</code> on each element	<code>ljust()</code>	Equivalent to <code>str.ljust()</code> on each element
<code>rjust()</code>	Equivalent to <code>str.rjust()</code> on each element	<code>center()</code>	Equivalent to <code>str.center()</code> on each element
<code>zfill()</code>	Equivalent to <code>str.zfill()</code> on each element	<code>strip()</code>	Equivalent to <code>str.strip()</code> on each element
<code>rstrip()</code>	Equivalent to <code>str.rstrip()</code> on each element	<code>lstrip()</code>	Equivalent to <code>str.lstrip()</code> on each element
<code>lower()</code>	Equivalent to <code>str.lower()</code> on each element	<code>upper()</code>	Equivalent to <code>str.upper()</code> on each element
<code>find()</code>	Equivalent to <code>str.find()</code> on each element	<code>rfind()</code>	Equivalent to <code>str.rfind()</code> on each element
<code>index()</code>	Equivalent to <code>str.index()</code> on each element	<code>rindex()</code>	Equivalent to <code>str.rindex()</code> on each element
<code>capitalize()</code>	Equivalent to <code>str.capitalize()</code> on each element	<code>swapcase()</code>	Equivalent to <code>str.swapcase()</code> on each element
<code>translate()</code>	Equivalent to <code>str.translate()</code> on each element	<code>startswith()</code>	Equivalent to <code>str.startswith()</code> on each element
<code>endswith()</code>	Equivalent to <code>str.endswith()</code> on each element	<code>isalnum()</code>	Equivalent to <code>str.isalnum()</code> on each element
<code>isalpha()</code>	Equivalent to <code>str.isalpha()</code> on each element	<code>isdigit()</code>	Equivalent to <code>str.isdigit()</code> on each element
<code>isspace()</code>	Equivalent to <code>str.isspace()</code> on each element	<code>istitle()</code>	Equivalent to <code>str.istitle()</code> on each element
<code>islower()</code>	Equivalent to <code>str.islower()</code> on each element	<code>isupper()</code>	Equivalent to <code>str.isupper()</code> on each element
<code>isnumeric()</code>	Equivalent to <code>str.isnumeric()</code> on each element	<code>isdecimal()</code>	Equivalent to <code>str.isdecimal()</code> on each element
<code>split()</code>	Equivalent to <code>str.split()</code> on each element	<code>rsplit()</code>	Equivalent to <code>str.rsplit()</code> on each element
<code>partition()</code>	Equivalent to <code>str.partition()</code> on each element	<code>rpartition()</code>	Equivalent to <code>str.rpartition()</code> on each element

Note that some of these, such as `capitalize()` above, return a series of strings:

```
monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                  'Eric Idle', 'Terry Jones', 'Michael Palin'])

monte.str.lower()
0    graham chapman
1      john cleese
2    terry gilliam
3       eric idle
```

```

4      terry jones
5      michael palin
dtype: object

```

others return numbers:

```

monte.str.len()
0      14
1      11
2      13
3       9
4      11
5      13
dtype: int64

```

others return boolean values:

```

monte.str.startswith('T')
0      False
1      False
2       True
3      False
4       True
5      False
dtype: bool

```

still others return lists or other compound values for each element:

```

monte.str.split()
0      [Graham, Chapman]
1      [John, Cleese]
2      [Terry, Gilliam]
3      [Eric, Idle]
4      [Terry, Jones]
5      [Michael, Palin]
dtype: object

```

The series-of-lists return value is the one that might give you pause: we'll take a look at this in more detail below.

Methods using Regular Expressions

In addition, there are several methods which accept regular expression to examine the content of each string element:

Method	Description	Method	Description
<code>match()</code>	Call <code>re.match()</code> on each element, returning a boolean.	<code>extract()</code>	Call <code>re.match()</code> on each element, returning matched groups as strings.
<code>findall()</code>	Call <code>re.findall()</code> on each element	<code>replace()</code>	Replace occurrences of pattern with some other string
<code>con tains()</code>	Call <code>re.search()</code> on each element, returning a boolean	<code>count()</code>	Count occurrences of pattern

Method	Description	Method	Description
<code>split()</code>	Equivalent to <code>str.split()</code> , but accepts regexps	<code>rsplit()</code>	Equivalent to <code>str.rsplit()</code> , but accepts regexps

With these, you can do a wide range of interesting operations. For example, we can extract the first name from each by asking for a contiguous group of characters at the beginning of each element:

```
monte.str.extract('([A-Za-z]+)')
0    Graham
1      John
2     Terry
3      Eric
4     Terry
5   Michael
dtype: object
```

Or we can do something more complicated, like finding all names which start and end with a consonant, making use of the the start-of-string (^) and end-of-string (\$) regular expression characters:

```
monte.str.findall(r'^[^AEIOU].*[^aeiou]$')
0    [Graham Chapman]
1          []
2    [Terry Gilliam]
3          []
4    [Terry Jones]
5  [Michael Palin]
dtype: object
```

The ability to apply regular expressions across Series or Dataframe entries opens up many possibilities for analysis and cleaning of data.

Miscellaneous Methods

Finally, there are some miscellaneous methods that enable other convenient operations:

Method	Description	Method	Description
<code>get()</code>	Index each element	<code>slice()</code>	Slice each element
<code>slice_replace()</code>	Replace slice in each element with passed value	<code>cat()</code>	Concatenate strings
<code>repeat()</code>	Repeat values	<code>normalize()</code>	Return unicode form of string
<code>pad()</code>	Add whitespace to left, right, or both sides of strings	<code>wrap()</code>	Split long strings into lines with length less than a given width

Method	Description	Method	Description
<code>join()</code>	Join strings in each element of the Series with passed separator	<code>get_dummies()</code>	extract dummy variables as a dataframe

Vectorized Item Access and Slicing. The `get()` and `slice()` operations, in particular, enable vectorized element access from each array. For example, we can get a slice of the first three characters of each array using `str.slice(0, 3)`. Note that this behavior is also available through Python’s normal indexing syntax; e.g. `df.str.slice(0, 3)` is equivalent to `df.str[0:3]`:

```
monte.str[0:3]
0    Gra
1    Joh
2    Ter
3    Eri
4    Ter
5    Mic
dtype: object
```

Indexing via `df.str.get(i)` and `df.str[i]` is likewise similar.

These `get()` and `slice()` methods also let you access elements of arrays returned by, e.g. `split()`. For example, to extract the last name of each entry, we can combine `split()` and `get()`:

```
monte.str.split().str.get(1)
0    Chapman
1    Cleese
2    Gilliam
3    Idle
4    Jones
5    Palin
dtype: object
```

Indicator Variables. Another method that requires a bit of extra explanation is the `get_dummies()` method. This is useful when your data has a column containing indicator variables. For example, we might have a dataset which contains information in the form of codes, such as A="born in America", B="born in the United Kingdom", C="likes cheese", D="likes spam":

```
full_monte = pd.DataFrame({'name': monte,
                           'info': ['B|C|D', 'B|D', 'A|C', 'B|D', 'B|C', 'B|C|D']})
full_monte
   info      name
0  B|C|D  Graham Chapman
1   B|D    John Cleese
2  A|C    Terry Gilliam
3  B|D      Eric Idle
```

```

4   B|C      Terry Jones
5  B|C|D    Michael Palin

```

The `get_dummies()` routine lets you quickly split-out these indicator variables into a dataframe:

```

full_monte['info'].str.get_dummies('|')

```

	A	B	C	D
0	0	1	1	1
1	0	1	0	1
2	1	0	1	0
3	0	1	0	1
4	0	1	1	0
5	0	1	1	1

With these operations as building blocks, you can construct an endless array of string processing procedures when cleaning your data.

Further Information

Above we saw just a brief survey of Pandas vectorized string methods. For more discussion of Python's string methods and basics of regular expressions, see Section X.X. For further examples of Pandas specialized string syntax, you can refer to Pandas' online documentation on [Working with Text Data](#).

Example: Recipe Database

These vectorized string operations become most useful in the process of cleaning-up messy real-world data. Here I'll walk through an example of that, using an open recipe database compiled from various sources on the web. Our goal will be to parse the recipe data into ingredient lists, so we can quickly find a recipe based on some ingredients we have on hand.

The scripts used to compile this can be found at <https://github.com/fictivekin/open-recipes>, and the link to the current version of the database is found there as well.

As of July 2015, this database is about 30 MB, and can be downloaded and unzipped with these commands:

```

# !curl -O http://openrecipes.s3.amazonaws.com/recipeitems-latest.json.gz
# !gunzip recipeitems-latest.json.gz

```

The database is in JSON format, so we will try `pd.read_json` to read it

```

try:
    recipes = pd.read_json('recipeitems-latest.json')
except ValueError as e:
    print("ValueError:", e)
ValueError: Trailing data

```


Oops! We get a `ValueError` mentioning that there is “trailing data”. Searching for this error on the internet, it seems that it’s due to using a file in which *each line* is itself a valid JSON, but the full file is not. Let’s check if this interpretation is true:

```
with open('recipeitems-latest.json') as f:
    line = f.readline()
pd.read_json(line).shape
(2, 12)
```

Yes, apparently each line is a valid JSON, so we’ll need to string them together. One way we can do this is to actually construct a string representation containing all these JSON entries, and then load the whole thing with `pd.read_json`:

```
# read the entire file into a python array
with open('recipeitems-latest.json', 'r') as f:
    # Extract each line
    data = (line.strip() for line in f)
    # Reformat so each line is the element of a list
    data_json = "[{}]" .format(', '.join(data))
# read the result as a JSON
recipes = pd.read_json(data_json)

recipes.shape
(173278, 17)
```

We see there are nearly 200,000 recipes, and we have 17 columns. Let’s take a look at one row to see what we have:

```
recipes.iloc[0]
_id                                {'$oid': '5160756b96cc62079cc2db15'}
cookTime                           PT30M
creator                             NaN
dateModified                        NaN
datePublished                       2013-03-11
description                        Late Saturday afternoon, after Marlboro Man ha...
image                              http://static.thepioneerwoman.com/cooking/file...
ingredients                        Biscuits\n3 cups All-purpose Flour\n2 Tablespo...
name                               Drop Biscuits and Sausage Gravy
prepTime                           PT10M
recipeCategory                      NaN
recipeInstructions                  NaN
recipeYield                         12
source                             thepioneerwoman
totalTime                           NaN
ts                                 {'$date': 1365276011104}
url                                http://thepioneerwoman.com/cooking/2013/03/dro...
Name: 0, dtype: object
```

There is a lot of information there, but much of it is in a very messy form, as is typical of data scraped from the web. In particular, the ingredient list is in string format; we’re going to have to carefully extract the information we’re interested in. Let’s start by taking a closer look at the ingredients:

```

recipes.ingredients.str.len().describe()
count    173278.000000
mean      244.617926
std       146.705285
min        0.000000
25%       147.000000
50%       221.000000
75%       314.000000
max       9067.000000
Name: ingredients, dtype: float64

```

The ingredient lists average 250 characters long, with a minimum of 0 and a maximum of nearly 10000 characters!

Just out of curiosity, let's see which recipe has the longest ingredient list:

```

recipes.name[np.argmax(recipes.ingredients.str.len())]
'Carrot Pineapple Spice & Brownie Layer Cake with Whipped Cream & Cream Cheese
Frosting and Marzipan Carrots'

```

That certainly looks like a complicated recipe.

We can do other aggregate explorations; for example, let's see how many of the recipes are for breakfast food:

```

recipes.description.str.contains('breakfast').sum()
3442

```

Or how many of the recipes list cinnamon as an ingredient:

```

recipes.ingredients.str.contains('[Cc]innamon').sum()
10526

```

We could even look to see whether any recipes mis-spell cinnamon with just a single “n”:

```

recipes.ingredients.str.contains('[Cc]inamon').sum()
11

```

This is the type of essential data exploration that is possible with Pandas string tools. It is data munging like this that Python really excels at.

Building a Recipe Recommender

Let's go a bit further, and start working on a simple recipe recommendation system. The task is straightforward: given a list of ingredients, find a recipe which uses all those ingredients. While conceptually straightforward, the task is complicated by the heterogeneity of the data: there is no easy operation, for example, to extract a clean list of ingredients from each row. So we will cheat a bit: we'll start with a list of common ingredients, and simply search to see whether they are in each recipe's ingredient list. For simplicity, let's just stick with herbs and spices for the time being:

```
spice_list = ['salt', 'pepper', 'oregano', 'sage', 'parsley',
              'rosemary', 'tarragon', 'thyme', 'paprika', 'cumin']
```

We can then build a boolean dataframe consisting of True and False values, indicating whether this ingredient appears in the list:

```
import re
spice_df = pd.DataFrame(dict((spice, recipes.ingredients.str.contains(spice,
re.IGNORECASE))
                             for spice in spice_list))
spice_df.head()
```

	cumin	oregano	paprika	parsley	pepper	rosemary	sage	salt	tarragon	thyme
0	False	False	False	False	False	False	True	False	False	False
1	False	False	False	False	False	False	False	False	False	False
2	True	False	False	False	True	False	False	True	False	False
3	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	False

Now, as an example, let's say we'd like to find a recipe which uses parsley, paprika, and tarragon. We can compute this very quickly using the `query()` method of dataframes, discussed in Section X.X:

```
selection = spice_df.query('parsley & paprika & tarragon')
len(selection)
10
```

We find only ten recipes with this combination; let's use the index returned by this selection to discover the names of the recipes which have this combination:

```
recipes.name[selection.index]
2069      All cremat with a Little Gem, dandelion and wa...
74964      Lobster with Thermidor butter
93768      Burton's Southern Fried Chicken with White Gravy
113926      Mijo's Slow Cooker Shredded Beef
137686      Asparagus Soup with Poached Eggs
140530      Fried Oyster Po'boys
158475      Lamb shank tagine with herb tabbouleh
158486      Southern fried chicken in buttermilk
163175      Fried Chicken Sliders with Pickles + Slaw
165243      Bar Tartine Cauliflower Salad
Name: name, dtype: object
```

Now that we have narrowed-down our recipe selection by a factor of almost 20,000, we are in a position to make a more informed decision about what recipe we'd like!

Going Further with Recipes

I hope the above example gives you a bit of a flavor (ha!) for the types of data cleaning operations that are efficiently enabled by Pandas' string methods. Of course, to build a very robust recipe recommendation system would require a *lot* more work! Extracting full ingredient lists from each recipe would be an important piece of the task; unfortunately the wide variety formats used makes this a very difficult and time-

consuming process. My goal with this example was not to do a complete, robust cleaning of the data, but to give you a taste (ha! again!) for how you might use these Pandas string tools in practice.

Working with Time Series

Pandas was developed in the context of financial modeling, so as you might expect, it contains a fairly extensive set of tools for working with dates, times, and time-indexed data. Date and time data comes in a few flavors, which we will discuss here:

- *Time stamps* reference particular moments in time; for example, July 4th, 2015 at 7:00am.
- *Time intervals* and *Periods* reference a length of time between a particular beginning and end point; for example, the year 2015. Periods usually reference a special case of time intervals in which each interval is of uniform length and does not overlap (for example, 24 hour-long periods in a day).
- *Time deltas* or *Durations* reference an exact length of time; for example, a duration of 22.56 seconds.

In this section we will introduce how to work with each of these types of date/time data in Pandas. This short section is by no means a complete guide to the timeseries tools available in Python or Pandas, but instead is intended as a broad overview of how you as a user should approach working with timeseries. We will start with a brief discussion of tools for dealing with dates and times in Python, before moving more specifically to a discussion of the tools provided by Pandas. After listing some resources that go into more depth, we will go through some short examples of working with timeseries data in Pandas.

Dates and Times in Python

The Python world has a number of available representations of dates, times, deltas, and timespans. While the time series tools provided by Pandas tend to be the most useful for data science applications, it is helpful to see their relationship to other packages used in Python.

Native Python Dates and Times: `datetime` and `dateutil`

Python's basic objects for working with dates and times reside in the built-in `date` `time` module. Along with the third-party `dateutil` module, you can use it to quickly perform a host of useful functionalities on dates and times. For example, you can manually build a date using the `datetime` type:

```
from datetime import datetime
datetime(year=2015, month=7, day=4)
datetime.datetime(2015, 7, 4, 0, 0)
```

Or, using the `dateutil` module, you can parse dates from a variety of string formats:

```
from dateutil import parser
date = parser.parse("4th of July, 2015")
date
datetime.datetime(2015, 7, 4, 0, 0)
```

Once you have a `datetime` object, you can do things like printing the day of the week:

```
date.strftime('%A')
'Saturday'
```

In the final line, we've used one of the standard string format codes for printing dates (`%A`), which you can read about in the [strftime section](#) of Python's [datetime documentation](#). Documentation of other useful date utilities can be found in [dateutil's online documentation](#). A related package to be aware of is [pytz](#), which contains tools for working with the most migraine-inducing piece of timeseries data: time zones.

The power of `datetime` and `dateutil` lie in their flexibility and easy syntax: you can use these objects and their built-in methods to easily perform nearly any operation you might be interested in. Where they break down is when you wish to work with large arrays of dates and times. Just as lists of Python numerical variables are suboptimal compared to NumPy-style typed numerical arrays, lists of Python `datetime` objects are suboptimal compared to typed arrays of encoded dates.

Typed arrays of times: NumPy's `datetime64`

The weaknesses of Python's `datetime` format inspired the NumPy team to add a set of native timeseries data type to NumPy. The `datetime64` dtype encodes dates as 64-bit integers, and thus allows arrays of dates to be represented very compactly. The `datetime64` requires a very specific input format:

```
import numpy as np
date = np.array('2015-07-04', dtype=np.datetime64)
date
array(datetime.date(2015, 7, 4), dtype='datetime64[D]')
```

Once we have this date formatted, however, we can quickly do vectorized operations on it:

```
date + np.arange(12)
array(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07',
      '2015-07-08', '2015-07-09', '2015-07-10', '2015-07-11',
      '2015-07-12', '2015-07-13', '2015-07-14', '2015-07-15'], dtype='datetime64[D]')
```

Because of the uniform type in NumPy `datetime64` arrays, this type of operation can be accomplished much more quickly than if we were working directly with Python's `datetime` objects, especially as arrays get large.

One detail of the `datetime64` and `timedelta64` objects is that they are built on a *fundamental time unit*. Because the `datetime64` object is limited to 64-bit precision, the range of encodable times is 2^{64} times this fundamental unit. In other words, `datetime64` imposes a tradeoff between *time resolution* and *maximum time span*.

For example, if you want a time resolution of one nanosecond, you only have enough information to encode a range of 2^{64} nanoseconds, or just under 600 years. NumPy will infer the desired unit from the input; for example, here is a day-based `datetime`:

```
np.datetime64('2015-07-04')
numpy.datetime64('2015-07-04')
```

Here is a minute-based `datetime`:

```
np.datetime64('2015-07-04 12:00')
numpy.datetime64('2015-07-04T12:00-0700')
```

(notice that the time zone is automatically set to the local time on the computer executing the code). You can force any desired fundamental unit using one of many format codes; for example, here we'll force a nanosecond-based time:

```
np.datetime64('2015-07-04 12:59:59.50', 'ns')
numpy.datetime64('2015-07-04T12:59:59.500000000-0700')
```

The following table, drawn from the NumPy `datetime64` documentation, lists the available format codes along with the relative and absolute timespans that they can encode:

Code	Meaning	Time span (relative)	Time span (absolute)
Y	year	$\pm 9.2e18$ years	[9.2e18 BC, 9.2e18 AD]
M	month	$\pm 7.6e17$ years	[7.6e17 BC, 7.6e17 AD]
W	week	$\pm 1.7e17$ years	[1.7e17 BC, 1.7e17 AD]
D	day	$\pm 2.5e16$ years	[2.5e16 BC, 2.5e16 AD]
h	hour	$\pm 1.0e15$ years	[1.0e15 BC, 1.0e15 AD]
m	minute	$\pm 1.7e13$ years	[1.7e13 BC, 1.7e13 AD]
s	second	$\pm 2.9e12$ years	[2.9e9 BC, 2.9e9 AD]
ms	millisecond	$\pm 2.9e9$ years	[2.9e6 BC, 2.9e6 AD]
us	microsecond	$\pm 2.9e6$ years	[290301 BC, 294241 AD]
ns	nanosecond	± 292 years	[1678 AD, 2262 AD]
ps	picosecond	± 106 days	[1969 AD, 1970 AD]
fs	femtosecond	± 2.6 hours	[1969 AD, 1970 AD]
as	attosecond	± 9.2 seconds	[1969 AD, 1970 AD]

For the types of data we see in the real world, a useful default is `datetime64[ns]`, as it can encode a useful range of modern dates with a suitably fine precision.

Finally, we will note that while the `datetime64` data type addresses some of the deficiencies of the built-in Python `datetime` type, it lacks many of the convenient methods and functions provided by `datetime` and especially `dateutil`. More information can be found in [NumPy's `datetime64` documentation](#).

Dates and Times in Pandas: Best of Both Worlds

Pandas builds upon all the above tools to provide a `Timestamp` object, which combines the ease-of-use of `datetime` and `dateutil` with the efficient storage and vectorized interface of `numpy.datetime64`. From a group of these `Timestamp` objects, Pandas can construct a `DatetimeIndex` which can be used to index data in a `Series` or `DataFrame`; we'll see many examples of this below.

For example, we can use Pandas tools to repeat the demonstration from above. We can parse a flexibly-formatted string date, and use format codes to output the day of the week:

```
import pandas as pd
date = pd.to_datetime("4th of July, 2015")
date
Timestamp('2015-07-04 00:00:00')

date.strftime('%A')
'Saturday'
```

Additionally, we can do NumPy-style vectorized operations directly on this same object:

```
date + pd.to_timedelta(np.arange(12), 'D')
DatetimeIndex(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07',
              '2015-07-08', '2015-07-09', '2015-07-10', '2015-07-11',
              '2015-07-12', '2015-07-13', '2015-07-14', '2015-07-15'],
              dtype='datetime64[ns]', freq=None, tz=None)
```

Below we will take a closer look at manipulating timeseries data with these tools provided by Pandas.

Pandas TimeSeries: Indexing by Time

Where the Pandas timeseries tools really become useful is when you begin to *index data by timestamps*. For example, we can construct a `Series` object which has time-indexed data:

```
index = pd.DatetimeIndex(['2014-07-04', '2014-08-04',
                          '2015-07-04', '2015-08-04'])
data = pd.Series([0, 1, 2, 3], index=index)
data
```

```
2014-07-04    0
2014-08-04    1
2015-07-04    2
2015-08-04    3
dtype: int64
```

Now that we have this data in a Series, we can make use of any of the Series indexing patterns we discussed in previous sections, passing values which can be coerced into dates:

```
data['2014-07-04':'2015-07-04']
2014-07-04    0
2014-08-04    1
2015-07-04    2
dtype: int64
```

There are additional special date-only indexing operations, such as passing a year to obtain a slice of all data from that year:

```
data['2015']
2015-07-04    2
2015-08-04    3
dtype: int64
```

We will see further examples below of the convenience of dates-as-indices. But first, a closer look at the available TimeSeries data structures.

Pandas TimeSeries Data Structures

This section will introduce the fundamental Pandas data structures for working with time series data.

- For *Time stamps*, Pandas provides the **Timestamp** type. As mentioned above, it is essentially a replacement for Python's native `datetime`, but is based on the more efficient `numpy.datetime64` data type. The associated Index structure is **DatetimeIndex**.
- For *Time Periods*, Pandas provides the **Period** type. This encodes a fixed-frequency interval based on `numpy.datetime64`. The associated index structure is **PeriodIndex**.
- For *Time deltas* or *Durations*, Pandas provides the **Timedelta** type. `Timedelta` is a more efficient replacement for Python's native `datetime.timedelta` type, and is based on `numpy.timedelta64`. The associated index structure is **TimedeltaIndex**.

The most fundamental of these date/time objects are the `Timestamp` and `DatetimeIndex` objects. While these class objects can be invoked directly, it is more common to use the `pd.to_datetime()` function, which can parse a wide variety of formats. Passing a single date to `pd.to_datetime()` yields a `Timestamp`; passing a series of dates by default yields a `DatetimeIndex`:


```

dates = pd.to_datetime([datetime(2015, 7, 3), '4th of July, 2015',
                        '2015-Jul-6', '07-07-2015', '20150708'])

dates
DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
               '2015-07-08'],
              dtype='datetime64[ns]', freq=None, tz=None)

```

Any DatetimeIndex can be converted to a PeriodIndex with the `to_period()` function with the addition of a frequency code; here we'll use 'D' to indicate daily frequency:

```

dates.to_period('D')
PeriodIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
            '2015-07-08'],
           dtype='int64', freq='D')

```

A TimedeltaIndex is created, for example, when a date is subtracted from another:

```

dates - dates[0]
TimedeltaIndex(['0 days', '1 days', '3 days', '4 days', '5 days'], dtype='timedelta64[ns]', freq=None)

```

Regular Sequences: `pd.date_range()`

To make the creation of regular date sequences more convenient, Pandas offers a few functions for this purpose: `pd.date_range()` for timestamps, `pd.period_range()` for periods, and `pd.timedelta_range()` for time deltas. We've seen that Python's `range()` and NumPy's `np.arange()` turn a startpoint, endpoint, and optional stepsize into a sequence. Similarly, `pd.date_range()` accepts a start date, an end date, and an optional frequency code to create a regular sequence of dates. By default, the frequency is one day:

```

pd.date_range('2015-07-03', '2015-07-10')
DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',
               '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],
              dtype='datetime64[ns]', freq='D', tz=None)

```

Optionally, the date range can be specified not with a start end end-point, but with a start-point and a number of periods:

```

pd.date_range('2015-07-03', periods=8)
DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',
               '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],
              dtype='datetime64[ns]', freq='D', tz=None)

```

The spacing can be modified by altering the `freq` argument, which defaults to D. For example, here we will construct a range of hourly timestamps:

```

pd.date_range('2015-07-03', periods=8, freq='H')
DatetimeIndex(['2015-07-03 00:00:00', '2015-07-03 01:00:00',
               '2015-07-03 02:00:00', '2015-07-03 03:00:00',
               '2015-07-03 04:00:00', '2015-07-03 05:00:00',

```

```
'2015-07-03 06:00:00', '2015-07-03 07:00:00'],
dtype='datetime64[ns]', freq='H', tz=None)
```

For more discussion of frequency options, see the [Frequency Codes](#) section below.

To create regular sequences of Period or Timedelta values, the very similar `pd.period_range()` and `pd.timedelta_range()` functions are useful. Here are some monthly periods:

```
pd.period_range('2015-07', periods=8, freq='M')
PeriodIndex(['2015-07', '2015-08', '2015-09', '2015-10', '2015-11', '2015-12',
            '2016-01', '2016-02'],
            dtype='int64', freq='M')
```

And a sequence of durations increasing by an hour:

```
pd.timedelta_range(0, periods=10, freq='H')
TimedeltaIndex(['00:00:00', '01:00:00', '02:00:00', '03:00:00', '04:00:00',
               '05:00:00', '06:00:00', '07:00:00', '08:00:00', '09:00:00'],
               dtype='timedelta64[ns]', freq='H')
```

All of these require understanding of Pandas frequency codes, which we'll summarize in the next section.

Frequencies and Offsets

Fundamental to these Pandas Timeseries tools is the concept of a frequency or date offset. Just as we saw the "D" (day) and "H" (hour) codes above, we can use such codes to specify any desired frequency spacing. Following is a summary of the main codes available:

Code	Description	Code	Description
D	calendar day	B	business day
W	weekly		
M	month end	BM	business month end
Q	quarter end	BQ	business quarter end
A	year end	BA	business year end
H	hours	BH	business hours
T	minutes		
S	seconds		
L	milliseonds		
U	microseconds		
N	nanoseconds	.	

The monthly, quarterly, annual frequencies are all marked at the end of the specified period. By adding an "S" suffix to any of these, they instead will be marked at the beginning:

Code	Description	Code	Description
MS	month start	BMS	business month start
QS	quarter start	BQS	business quarter start
AS	year start	BAS	business year start

Additionally, you can change the month used to mark any quarterly or annual code by adding a three-letter month code as a suffix:

- Q-JAN, BQ-FEB, QS-MAR, BQS-APR, etc.
- A-JAN, BA-FEB, AS-MAR, BAS-APR, etc.

In the same way, the split-point of the weekly frequency can be modified by adding a three-letter weekday code:

- W-SUN, W-MON, W-TUE, W-WED, etc.

On top of this, codes can be combined with numbers to specify other frequencies. For example, for a frequency of 2 hours 30 minutes, we can combine the hour ("H") and minute ("T") codes as follows:

```
pd.timedelta_range(0, periods=9, freq="2H30T")
TimedeltaIndex(['00:00:00', '02:30:00', '05:00:00', '07:30:00', '10:00:00',
               '12:30:00', '15:00:00', '17:30:00', '20:00:00'],
              dtype='timedelta64[ns]', freq='150T')
```

All of these short codes refer to specific instances of Pandas time series offsets, which can be found in the `pd.tseries.offsets` module. For example, we can create a business day offset directly as follows:

```
from pandas.tseries.offsets import BDay
pd.date_range('2015-07-01', periods=5, freq=BDay())
DatetimeIndex(['2015-07-01', '2015-07-02', '2015-07-03', '2015-07-06',
              '2015-07-07'],
             dtype='datetime64[ns]', freq='B', tz=None)
```

For more discussion of the use of frequencies and offsets, see the Pandas online [DateOffset documentation](#).

Resampling, Shifting, and Windowing

The ability to use dates and times as indices to intuitively organize and access data is an important piece of the Pandas time series tools. The benefits of indexed data in

general (automatic alignment during operations, intuitive data slicing and access, etc.) certainly apply, but Pandas also provides several timeseries-specific operations.

We will take a look at a few of those here, using some stock price data as an example. Because Pandas was developed largely in a finance context, it includes some very specific tools for financial data. For example, Pandas has built-in tool for reading available financial indices, the `DataReader` function. This function knows how to import financial data from a number of available sources, including Yahoo finance, Google Finance, and others. Here we will load Google's closing price history using Pandas:

```
from pandas.io.data import DataReader

goog = DataReader('GOOG', start='2004', end='2015',
                  data_source='google')
goog.head()
```

	Open	High	Low	Close	Volume
Date					
2004-08-19	49.96	51.98	47.93	50.12	NaN
2004-08-20	50.69	54.49	50.20	54.10	NaN
2004-08-23	55.32	56.68	54.47	54.65	NaN
2004-08-24	55.56	55.74	51.73	52.38	NaN
2004-08-25	52.43	53.95	51.89	52.95	NaN

for simplicity, we'll use just the closing price:

```
goog = goog['Close']
```

We can visualize this using the `plot()` method, after the normal matplotlib setup boilerplate:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
goog.plot();
```

Resampling and Converting Frequencies

One common need for time series data is resampling at a higher or lower frequency. This can be done using the `resample()` method, or the much simpler `asfreq()` method. The primary difference between the two is that `resample()` is fundamentally a *data aggregation*, while `asfreq()` is fundamentally a *data selection*.

Taking a look at the google closing price, let's compare what the two return when we down-sample the data. Here we will resample the data at the end of business year:

```
goog.plot(alpha=0.5)
goog.resample('BA', how='mean').plot()
goog.asfreq('BA').plot();
plt.legend(['input', 'resample', 'asfreq'],
           loc='upper left');
```

Notice the difference: at each point, `resample` reports the *average of the previous year*, while `asfreq` reports the *value at the end of the year*.

For up-sampling, `resample()` and `asfreq()` are largely equivalent, though `resample` has many more options available. In this case, the default for both methods is to leave the up-sampled points empty, that is, filled with NA values. Just as with the `pd.fillna()` function discussed previously, `asfreq()` accepts a `method` argument to specify how values are imputed. Here, we will resample the business day data at a daily frequency (i.e. including weekends):

```
fig, ax = plt.subplots(2, sharex=True)
data = goog.iloc[:10]

data.asfreq('D').plot(ax=ax[0], marker='o')

data.asfreq('D', method='bfill').plot(ax=ax[1], marker='o')
data.asfreq('D', method='ffill').plot(ax=ax[1], marker='o')
ax[1].legend(["back-fill", "forward-fill"]);
```

The top panel is the default: non-business days are left as NA values and do not appear on the plot. The bottom panel shows the differences between two strategies for filling the gaps: forward-filling and backward-filling.

Time-shifts

Another common timeseries-specific operation is shifting of data in time. Pandas has two closely-related methods for computing this: `shift()` and `tshift()`. In short, the difference between them is that `shift()` *shifts the data*, while `tshift()` *shifts the index*. In both cases, the shift is specified in multiples of the frequency.

Here we will both `shift()` and `tshift()` by 1000 days;

```
fig, ax = plt.subplots(3, sharex=True)

# apply a frequency to the data
goog = goog.asfreq('D', method='pad')

goog.plot(ax=ax[0])
ax[0].legend(['input'], loc=2)

goog.shift(900).plot(ax=ax[1])
ax[1].legend(['shift(900)'], loc=2)

goog.tshift(900).plot(ax=ax[2])
ax[2].legend(["tshift(900)"], loc=2);
```

We see here visually that the `shift(900)` shifts the data by 900 days, pushing some of it off the end of the graph (and leaving NA values at the other end). On the other hand, carefully examining the x labels, we see that `tshift(900)` leaves the data in place while shifting the time index itself by 900 days.

A common context for this type of shift is in computing differences over time. For example, we use shifted values to compute the one-year return on investment for Google stock over the course of the dataset:

```
ROI = 100 * (goog.tshift(-365) / goog - 1)
ROI.plot()
plt.ylabel('% Return on Investment');
```

This helps us to see the overall trend in Google stock: thus far, the most profitable times to invest in Google have been (unsurprisingly, in retrospect) shortly after its IPO, and in the middle of the 2009 recession.

Rolling Windows

Rolling statistics are a third type of timeseries-specific operation implemented by Pandas. These can be accomplished via one of several functions, such as `pd.rolling_mean()`, `pd.rolling_sum()`, `pd.rolling_min()`, etc. Just about every pandas aggregation function (see Section X.X) has an associated rolling function.

The syntax of all of these is very similar: for example, here is the one-year centered rolling mean of the Google stock prices:

```
rmean = pd.rolling_mean(goog, 365, freq='D', center=True)
rstd = pd.rolling_std(goog, 365, freq='D', center=True)

data = pd.DataFrame({'input': goog, 'one-year rolling_mean': rmean, 'one-year
rolling_std': rstd})
ax = data.plot()
ax.lines[0].set_alpha(0.3)
```

Along with the rolling versions of standard aggregates, there are also the more flexible functions `pd.rolling_window()` and `pd.rolling_apply()`. For details, see the documentation of these functions, or the example below.

Where to Learn More

The above is only a brief summary of some of the most essential features of time series tools provided by Pandas; for a more complete discussion you can refer to [Pandas Time Series Documentation](#).

Another excellent resource is the textbook [Python for Data Analysis](#) by Wes McKinney (O'Reilly, 2012). Though it is now a few years old, it is an invaluable resource on the use of Pandas. In particular, this book emphasizes time series tools in the context of business and finance, and focuses much more on particular details of business calendars, time zones, and related topics.

As usual, you can also use the IPython help functionality to explore and try further options available to the functions and methods discussed above: I find this often is the best way to learn a new Python tool.

Example: Visualizing Seattle Bicycle Counts

As a more involved example of working with some time series data, let's take a look at bicycle counts on Seattle's **Fremont Bridge**. This data comes from an automated bicycle counter, installed in late 2012, which has inductive sensors on the east and west sidewalks of the bridge. The hourly bicycle counts can be downloaded from <http://data.seattle.gov/>; here is the [direct link to the dataset](#).

As of summer 2015, the CSV can be downloaded as follows:

```
# !curl -o FremontBridge.csv https://data.seattle.gov/api/views/65db-xm6k/rows.csv?accessType=DOWNLOAD
```

Once this dataset is downloaded, we can use Pandas to read the CSV output into a dataframe. We will specify that we want the Date as an index, and we want these dates to be automatically parsed:

```
data = pd.read_csv('FremontBridge.csv', index_col='Date', parse_dates=True)
data.head()
```

```

                Fremont Bridge West Sidewalk \
Date
2012-10-03 00:00:00                        4
2012-10-03 01:00:00                        4
2012-10-03 02:00:00                        1
2012-10-03 03:00:00                        2
2012-10-03 04:00:00                        6
```

```

                Fremont Bridge East Sidewalk
Date
2012-10-03 00:00:00                        9
2012-10-03 01:00:00                        6
2012-10-03 02:00:00                        1
2012-10-03 03:00:00                        3
2012-10-03 04:00:00                        1
```

For convenience, we'll further process this dataset by shortening the column names and adding a "Total" column:

```
data.columns = ['West', 'East']
data['Total'] = data.eval('West + East')
```

Now let's take a look at the summary statistics for this data:

```
data.describe()
```

	West	East	Total
count	24017.000000	24017.000000	24017.000000
mean	55.452180	52.088646	107.540825
std	70.721848	74.615127	131.327728
min	0.000000	0.000000	0.000000
25%	7.000000	7.000000	16.000000
50%	31.000000	27.000000	62.000000
75%	74.000000	65.000000	143.000000
max	698.000000	667.000000	946.000000

Visualizing the Data

We can gain some insight into the dataset by visualizing it. Let's start by plotting the raw data:

```
%matplotlib inline
import seaborn; seaborn.set()

data.plot()
plt.ylabel('Hourly Bicycle Count');
```

The ~25000 hourly samples are far too dense for us to make much sense of. We can gain more insight by resampling the data to a coarser grid. Let's resample by week:

```
data.resample('W', how='sum').plot()
plt.ylabel('Weekly bicycle count');
```

This shows us some interesting seasonal trends: as you might expect, people bicycle more in the summer than in the winter, and even within a particular season the bicycle use varies from week to week (likely dependent on weather; see Section X.X where we explore this further).

Another useful way to aggregate the data is to use a rolling mean, using the `pd.rolling_mean()` function. Here we'll do a 30 day rolling mean of our data, making sure to center the window:

```
pd.rolling_mean(data, 30, freq='D', center=True).plot()
plt.ylabel('mean hourly count');
```

The jaggedness of the result is due to the hard cutoff of the window. We can get a smoother version of a rolling mean using a window function, for example, a Gaussian window. Here we need to specify both the width of the window (we choose 50 days) and the width of the Gaussian within the window (we choose 10 days):

```
pd.rolling_window(data, 50, freq='D', center=True,
                  win_type='gaussian', std=10).plot()
plt.ylabel('smoothed hourly count')
```

Digging Into the Data

While these smoothed data views are useful to get an idea of the general trend in the data, they hide much of the interesting structure. For example, we might want to look at the average traffic as a function of the time of day. We can do this using the `GroupBy` functionality discussed in Section X.X:

```
by_time = data.groupby(data.index.time).mean()
hourly_ticks = 4 * 60 * 60 * np.arange(6)
by_time.plot(xticks=hourly_ticks);
```

The hourly traffic is a strongly bimodal distribution, with peaks around 8:00 in the morning and 5:00 in the evening. This is likely evidence of a strong component of commuter traffic crossing the bridge. This is further evidenced by the differences

between the western sidewalk (generally used going toward downtown Seattle), which peaks more strongly in the morning, and the eastern sidewalk (generally used going away from downtown Seattle), which peaks more strongly in the evening.

We also might be curious about how things change based on the day of the week. Again, we can do this with a simple groupby:

```
by_weekday = data.groupby(data.index.dayofweek).mean()
by_weekday.index = ['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun']
by_weekday.plot();
```

This shows a strong distinction between weekday and weekend totals, with around twice as many average riders crossing the bridge on Monday-Friday than on Saturday and Sunday.

With this in mind, let's do a compound GroupBy and look at the hourly trend on weekdays vs weekends. We'll start by grouping by both a flag marking the weekend, and the time of day:

```
weekend = np.where(data.index.weekday < 5, 'Weekday', 'Weekend')
by_time = data.groupby([weekend, data.index.time]).mean()
```

Now we'll use some of the matplotlib tools described in Section X.X to plot two panels side-by-side:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, 2, figsize=(14, 5))
by_time.ix['Weekday'].plot(ax=ax[0], title='Weekdays', xticks=hourly_ticks)
by_time.ix['Weekend'].plot(ax=ax[1], title='Weekends', xticks=hourly_ticks);
```

The result is very interesting: we see a bimodal commute pattern during the work week, and a unimodal recreational pattern during the weekends. It would be interesting to dig through this data in more detail, and examine the effect of weather, temperature, time of year, etc. on people's commuting patterns. I did this a bit in a blog post using a subset of this data; you can find that discussion [on my blog](#). We will also revisit this dataset in the context of modeling in Section X.X.

High-Performance Pandas: `eval()` and `query()`

As we've seen in the previous chapters, the power of the PyData stack lies in the ability of NumPy and Pandas to push basic operations into C via an intuitive syntax: examples are vectorized/broadcasted operations in NumPy, and grouping-type operations in Pandas. While these abstractions are efficient and effective for many common use-cases, they often rely on the creation of temporary intermediate objects which can cause undue overhead in computational time and memory use. Many of the Python performance solutions explored in chapter X.X are designed to address these deficiencies, and we'll explore these in more detail at that point.

As of version 0.13 (released January 2014), Pandas includes some experimental tools which allow you to directly access C-speed operations without costly allocation of intermediate arrays. These are the `eval()` and `query()` functions, which rely on the `numexpr` package (discussed more fully in section X.X). In this notebook we will walk through their use and give some rules-of-thumb about when you might think about using them.

Motivating `query()` and `eval()`: Compound Expressions

We've seen previously that NumPy and Pandas support fast vectorized operations; for example, when adding the elements of two arrays:

```
import numpy as np
rng = np.random.RandomState(42)
x = rng.rand(1E6)
y = rng.rand(1E6)
%timeit x + y
100 loops, best of 3: 3.57 ms per loop
```

As discussed in Section X.X, this is much faster than doing the addition via a Python loop or comprehension

```
%timeit np.fromiter((xi + yi for xi, yi in zip(x, y)), dtype=x.dtype,
count=len(x))
1 loops, best of 3: 232 ms per loop
```

But this abstraction can become less efficient when computing compound expressions. For example, consider the following expression:

```
mask = (x > 0.5) & (y < 0.5)
```

Because NumPy evaluates each subexpression, this is roughly equivalent to the following

```
tmp1 = (x > 0.5)
tmp2 = (y < 0.5)
mask = tmp1 & tmp2
```

In other words, *every intermediate step is explicitly allocated in memory*. If the `x` and `y` arrays are very large, this can lead to significant memory and computational overhead. The `numexpr` library gives you the ability to compute this type of compound expression element-by-element, without the need to allocate full intermediate arrays. More details on `numexpr` are given in section X.X, but for the time being it is sufficient to say that the library accepts a *string* giving the NumPy-style expression you'd like to compute:

```
import numexpr
mask_numexpr = numexpr.evaluate('(x > 0.5) & (y < 0.5)')
np.allclose(mask, mask_numexpr)
True
```

The benefit here is that NumExpr evaluates the expression in a way that does not use full-sized temporary arrays, and thus can be much more efficient than NumPy, especially for large arrays.

The Pandas `eval()` and `query()` tools discussed below are conceptually similar, and depend on the numexpr package.

pandas.eval() for Efficient Operations

The `eval()` function in Pandas uses string expressions to efficiently compute operations using dataframes. For example, consider the following dataframes:

```
import pandas as pd
nrows, ncols = 100000, 100
rng = np.random.RandomState(42)
df1, df2, df3, df4 = (pd.DataFrame(rng.rand(nrows, ncols))
                       for i in range(4))
```

To compute the sum of all four dataframes using the typical Pandas approach, we can just write the sum:

```
%timeit df1 + df2 + df3 + df4
10 loops, best of 3: 88.6 ms per loop
```

The same result can be computed via `pd.eval` by constructing the expression as a string:

```
%timeit pd.eval('df1 + df2 + df3 + df4')
10 loops, best of 3: 42.4 ms per loop
```

The `eval()` version of this expression is about 50% faster (and uses much less memory), while giving the same result:

```
np.allclose(df1 + df2 + df3 + df4,
            pd.eval('df1 + df2 + df3 + df4'))
True
```

Operations Supported by `pd.eval()`

As of Pandas v0.16, `pd.eval()` supports a wide range of operations. To demonstrate these, we'll use the following integer dataframes:

```
df1, df2, df3, df4, df5 = (pd.DataFrame(rng.randint(0, 1000, (100, 3)))
                             for i in range(5))
```

Arithmetic Operators. `pd.eval()` supports all arithmetic operators; e.g.

```
result1 = -df1 * df2 / (df3 + df4) - df5
result2 = pd.eval('-df1 * df2 / (df3 + df4) - df5')
np.allclose(result1, result2)
True
```

Comparison Operators. `pd.eval()` supports all comparison operators, including chained expressions; e.g.

```
result1 = (df1 < df2) & (df2 <= df3) & (df3 != df4)
result2 = pd.eval('df1 < df2 <= df3 != df4')
np.allclose(result1, result2)
True
```

Bitwise Operators. `pd.eval()` supports the `&` and `|` bitwise operators:

```
result1 = (df1 < 0.5) & (df2 < 0.5) | (df3 < df4)
result2 = pd.eval('(df1 < 0.5) & (df2 < 0.5) | (df3 < df4)')
np.allclose(result1, result2)
True
```

In addition, it supports the use of the literal `and` and `or` in boolean expressions:

```
result3 = pd.eval('(df1 < 0.5) and (df2 < 0.5) or (df3 < df4)')
np.allclose(result1, result3)
True
```

Object Attributes and indices. `pd.eval()` supports access to object attributes via the `obj.attr` syntax, and indexes via the `obj[index]` syntax:

```
result1 = df2.T[0] + df3.iloc[1]
result2 = pd.eval('df2.T[0] + df3.iloc[1]')
np.allclose(result1, result2)
True
```

Other Operations. Other operations such as function calls, conditional statements, loops, and other more involved constructs are currently **not** implemented in `pd.eval()`. If you'd like to execute these more complicated types of expressions, you can use the `numexpr` library itself, discussed in Section X.X.

DataFrame.eval() for Column-wise Operations

Just as Pandas has a top-level `pd.eval()` function, DataFrames have an `eval()` method that works in similar ways. The benefit of the `eval()` method is that columns can be referred to **by name**. We'll use this labeled array as an example:

```
df = pd.DataFrame(rng.rand(1000, 3), columns=['A', 'B', 'C'])
df.head()
```

	A	B	C
0	0.375506	0.406939	0.069938
1	0.069087	0.235615	0.154374
2	0.677945	0.433839	0.652324
3	0.264038	0.808055	0.347197
4	0.589161	0.252418	0.557789

Using `pd.eval()` as above, we can compute expressions with the three columns like this:

```
result1 = (df['A'] + df['B']) / (df['C'] - 1)
result2 = pd.eval("(df.A + df.B) / (df.C - 1)")
np.allclose(result1, result2)
True
```

The DataFrame `eval()` method allows much more succinct evaluation of expressions with the columns:

```
result3 = df.eval('(A + B) / (C - 1)')
np.allclose(result1, result3)
True
```

Notice here that we treat *column names as variables* within the evaluated expression, and the result is what we would wish.

Assignment in `DataFrame.eval()`

In addition to the options discussed above, `DataFrame.eval()` also allows assignment to any column. Let's use the dataframe from above, which has columns 'A', 'B', and 'C':

```
df.head()
   A      B      C
0  0.375506  0.406939  0.069938
1  0.069087  0.235615  0.154374
2  0.677945  0.433839  0.652324
3  0.264038  0.808055  0.347197
4  0.589161  0.252418  0.557789
```

We can use `df.eval()` to create a new column 'D' and assign to it a value computed from the other columns:

```
df.eval('D = (A + B) / C')
df.head()
   A      B      C      D
0  0.375506  0.406939  0.069938  11.187620
1  0.069087  0.235615  0.154374   1.973796
2  0.677945  0.433839  0.652324   1.704344
3  0.264038  0.808055  0.347197   3.087857
4  0.589161  0.252418  0.557789   1.508776
```

In the same way, any existing column can be modified:

```
df.eval('D = (A - B) / C')
df.head()
   A      B      C      D
0  0.375506  0.406939  0.069938 -0.449425
1  0.069087  0.235615  0.154374 -1.078728
2  0.677945  0.433839  0.652324  0.374209
```

```
3  0.264038  0.808055  0.347197 -1.566886
4  0.589161  0.252418  0.557789  0.603708
```

Local Variables in DataFrame.eval()

The `DataFrame.eval()` method supports an additional syntax which lets it work with local Python variables. Consider the following:

```
column_mean = df.mean(1)
result1 = df['A'] + column_mean
result2 = df.eval('A + @column_mean')
np.allclose(result1, result2)
True
```

The `@` character here marks a *variable name* rather than a *column name*, and lets you efficiently evaluate expressions involving the two “namespaces”: the namespace of columns, and the namespace of Python objects. Notice that this `@` character is only supported by the `DataFrame.eval()` *method*, not by the `pandas.eval()` *function*, because the `pandas.eval()` function only has access to the one (Python) namespace.

DataFrame.query() Method

The dataframe has another method based on evaluated strings, called the `query()` method. Consider the following:

```
result1 = df[(df.A < 0.5) & (df.B < 0.5)]
result2 = pd.eval('df[(df.A < 0.5) & (df.B < 0.5)]')
np.allclose(result1, result2)
True
```

As with the example in `DataFrame.eval()` above, this is an expression involving columns of the dataframe. It cannot, however, be expressed using the `DataFrame.eval()` syntax! Instead, for this type of filtering operation, you can use the `query()` method:

```
result2 = df.query('A < 0.5 and B < 0.5')
np.allclose(result1, result2)
True
```

In addition to being a more efficient computation, compared to the masking expression this is much easier to read and understand. Note that the `query()` method also accepts the `@` flag to mark local variables:

```
Cmean = df['C'].mean()
result1 = df[(df.A < Cmean) & (df.B < Cmean)]
result2 = df.query('A < @Cmean and B < @Cmean')
np.allclose(result1, result2)
True
```

Performance: When to Use these functions

This is all well and good, but when should you use this functionality? There are two considerations here: *computation time* and *memory use*.

Memory use is the most predictable aspect. As mentioned above, every compound expression involving NumPy arrays or Pandas DataFrames will result in implicit creation of temporary arrays: For example, this:

```
x = df[(df.A < 0.5) & (df.B < 0.5)]
```

Is roughly equivalent to this:

```
tmp1 = df.A < 0.5
tmp2 = df.B < 0.5
tmp3 = tmp1 & tmp2
x = df[tmp3]
```

If the size of the temporary dataframes is significant compared to your available system memory (typically a few gigabytes in 2015) then it's a good idea to use an `eval()` or `query()` expression. You can check the approximate size of your array in bytes using, e.g.

```
df.values.nbytes
32000
```

On the performance side, `eval()` can be faster even when you are not maxing-out your system memory. The issue is how your temporary dataframes compare to the size of the L1 or L2 CPU cache on your system (typically a few megabytes in 2015); if they are much bigger then `eval()` can avoid some potentially slow movement of values between the different memory caches.

For a simple benchmark of these two, we can use `%timeit` to see the performance breakdown. The following cells will take a few minutes to run, as they automatically repeat the calculations in order to remove system timing variations.

```
sizes = (10 ** np.linspace(3, 7, 7)).astype(int)
times_eval_p = np.zeros_like(sizes, dtype=float)
times_eval = np.zeros_like(sizes, dtype=float)
x = rng.rand(4, max(sizes), 2)

for i, size in enumerate(sizes):
    rng = np.random.RandomState(0)
    df1, df2, df3, df4 = (pd.DataFrame(x[i, :size])
                          for i in range(4))
    t = %timeit -oq df1 + df2 + df3 + df4
    times_eval_p[i] = t.best
    t = %timeit -oq pd.eval('df1 + df2 + df3 + df4')
    times_eval[i] = t.best

times_query_p = np.zeros_like(sizes, dtype=float)
times_query = np.zeros_like(sizes, dtype=float)
```

```

x = rng.rand(max(sizes), 2)

for i, size in enumerate(sizes):
    rng = np.random.RandomState(0)
    df1 = pd.DataFrame(x[:size], columns=['A', 'B'])
    t = %timeit -oq df1[(df1.A < 0.5) & (df1.B < 0.5)]
    times_query_p[i] = t.best
    t = %timeit -oq df1.query('(A < 0.5) & (B < 0.5)')
    times_query[i] = t.best

%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
fig, ax = plt.subplots(1, 2, figsize=(10, 4))
ax[0].loglog(2 * sizes, times_eval_p, label='python')
ax[0].loglog(2 * sizes, times_eval, label='eval')
ax[0].legend(loc='upper left')
ax[0].set(xlabel='# array elements',
          ylabel='query execution time (s)',
          title='pd.eval("df1 + df2 + df3 + df4")')

ax[1].loglog(2 * sizes, times_query_p, label='python')
ax[1].loglog(2 * sizes, times_query, label='eval')
ax[1].legend(loc='upper left')
ax[1].set(xlabel='# array elements',
          ylabel='query execution time (s)',
          title='DataFrame.query("(A < 0.5) & (B < 0.5)")');

```

We see that on my machine, the simple Python expression is faster for arrays with fewer than roughly 10^5 or 10^6 elements (though it will use more memory), and the `eval()`/`query()` approach is faster for arrays much larger than this. Keep this in mind as you decide how to optimize your own code!

Learning More

We've covered most of the details of `eval()` and `query()` here; for more information on these you can refer to the Pandas documentation. In particular, different parsers and engines can be specified for running these queries; for details on this see the discussion within the [Enhancing Performance](#) section of the documentation. Regarding more general use of this numexpr string-to-compiled-code interface, refer to Section X.X, where we discuss the NumExpr package in more detail.

Further Resources

In this chapter we've covered many of the basics of using Pandas effectively for data analysis. Still, much has been left out. To learn more about Pandas, I recommend some of the following resources:

- **Pandas Online Documentation:** this is the go-to source for complete documentation of the package. While the examples in the documentation tend to be small generated datasets, the description of the options is complete and generally very useful for understanding the use of various functions.
- ***Python for Data Analysis*** by Wes McKinney (O'Reilly, 2012). Wes is the creator of Pandas, and his book contains much more detail on the Pandas package than we had room for in this chapter. In particular, he takes a deep dive into tools for TimeSeries, which were his bread and butter as a financial consultant. The book also has many entertaining examples of applying Pandas to gain insight from real-world datasets. Keep in mind, though, that the book is now several years old, and the Pandas package has quite a few new features that this book does not cover.
- **StackOverflow Pandas:** Pandas has so many users that any question you have has likely been asked and answered on StackOverflow. Using Pandas is a case where some Google-Fu is your best friend. Simply type-in to your favorite search engine the question, problem, or error you're coming across, and more than likely you'll find your answer on a StackOverflow page.
- **Pandas on PyVideo:** From PyCon to SciPy to PyData, many conferences have featured Pandas tutorials from Pandas developers and power-users. The PyCon tutorials in particular tend to be given by very well-vetted presenters.

Using the above resources, combined with the walk-through given in this section, my hope is that you'll be poised to use Pandas to tackle any data analysis problem you come across!